OIPE IAP29 DEC 02 2005 PATENT & TRADEMARK OFFICE

AF/2154 IFW

PTO/SB/21 (02-04)
Approved for use through 07/31/2006. OMB 0651-0031
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Application Number	09/881,915
Filing Date	06/14/2001
First Named Inventor	William K. Bodin
Art Unit	2154
Examiner Name	Patel, Haresh N.
Attorney Docket Number	AUS920010502US1
Total Number of Pages in This Submission	46

ENCLOSURES *(Check all that apply)*

- ☐ Fee Transmittal Form
  - ☐ Fee Attached
- ☐ Amendment/Reply
  - ☐ After Final
  - ☐ Affidavits/declaration(s)
- ☐ Extension of Time Request
- ☐ Express Abandonment Request
- ☐ Information Disclosure Statement
- ☐ Certified Copy of Priority Document(s)
- ☐ Response to Missing Parts/ Incomplete Application
  - ☐ Response to Missing Parts under 37 CFR 1.52 or 1.53
- ☐ Drawing(s)
- ☐ Licensing-related Papers
- ☐ Petition
- ☐ Petition to Convert to a Provisional Application
- ☐ Power of Attorney, Revocation Change of Correspondence Address
- ☐ Terminal Disclaimer
- ☐ Request for Refund
- ☐ CD, Number of CD(s) ____________
- ☐ After Allowance communication to Technology Center (TC)
- ☐ Appeal Communication to Board of Appeals and Interferences
- ☑ Appeal Communication to TC **(Appeal Notice, Brief, Reply Brief)**
- ☐ Proprietary Information
- ☐ Status Letter
- ☑ Other Enclosure(s) (please Identify below):

Appeal Brief; 3 References; and Postcard

Remarks

The Comissioner is authorized to charge or credit Deposit Account No. 09-0447.

Customer No. 34533.

SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm or Individual name	John Biggers Reg. No. 44,537
Signature	John Biggers
Date	November 30, 2005

CERTIFICATE OF TRANSMISSION/MAILING

I hereby certify that this correspondence is being facsimile transmitted to the USPTO or deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below.

Typed or printed name	Catherine Berglund		
Signature	Catherine Berglund	Date	November 30, 2005

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: William K. Bodin, *et al.*	§	Group Art Unit: 2154
Serial No.: 09/881,915	§	Examiner: Patel, Haresh N.
Filed: June 14, 1002	§	Atty Docket No.: AUS920010502US1
Title: Streaming Digital Content Under Remote Direction	§	

Mail Stop: Appeal Brief-Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450

CERTIFICATE OF TRANSMISSION/MAILING
I hereby certify that this correspondence is being facsimile transmitted to the USPTO at 571-273-8300 or deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on this date:
November 30, 2005
Date
Catherine Berglund
Catherine Berglund

APPEAL BRIEF

Honorable Commissioner:

This is an Appeal Brief filed pursuant to 37 CFR § 41.37 in response to the Final Office Action of June 30, 2005 ('Final Office Action'), and pursuant to the Notice of Appeal filed September 30, 2005.

REAL PARTY IN INTEREST

The real party in interest is the patent assignee, International Business Machines Corporation ("IBM"), a New York corporation having a place of business at Armonk, New York 10504.

RELATED APPEALS AND INTERFERENCES

There are no related appeals or interferences.

STATUS OF CLAIMS

Claims 1-36 are pending in the case. All pending claims are on appeal.

STATUS OF AMENDMENTS

No amendments were submitted after final rejection. The claims as currently presented are included in the Appendix of Claims that accompanies this Appeal Brief.

SUMMARY OF CLAIMED SUBJECT MATTER

Applicants provide the following concise summary of the claimed subject matter according to 37 CFR § 41.37(c)(1)(vii), including references to specification by page and line number and to the drawing(s) if any, by reference characters.

Methods, systems, and computer program products are provided for remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices the method implemented upon a network of digital computers, at least one of the digital computers comprising a content server upon which the steps of the method are implemented in computer memory and at least one computer processor (described for example at reference numbers 100, 102, 106, and 108 in Figure 1 and lines 5-9 on page 13). Embodiments of methods typically include receiving digital content from the sources, the digital content having a multiplicity of digital formats (described for example at reference numbers 106 and 221 in Figure 2 and lines 11-17 on page 13); receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream (described for example at references

numbers 104, 202, and 204 in Figure 2 and lines 19-22 on page 13); transcoding the digital content from sources into digital content having streaming format (described for example at references 220 and 223 and lines 22-23 on page 13); including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format (described for example at reference 225 in Figure 2 and lines 25-27 on page 13); and communicating to at least one of the client devices the output stream (described for example at references 102 and 225 in Figure 2 and line 27 on page 13 through line 2 on page 14). In typical embodiments, the client devices comprise client device attributes, said transcoding further comprising transcoding in dependence upon the client device attributes (described for example at lines 4-7 on page 14 and lines 11-17 on page 14). In typical embodiments, client device attributes include device type, screen size, frame rate, and audio status (described for example at lines 7-9 on page 14).

In typical embodiments, a remote director comprises a personal computer coupled through a network to a content server, and embodiments of inventive methods typically include sending from the remote director to the content server remote director instructions (described for example at references 100, 104, and 204 in Figure 2 and line 19 on page 14 through line 3 on page 15). In typical embodiments, a hyperlinked URL invoked through a hot spot on a video screen of a remote director in turn invokes on a content server a servlet (described for example at lines 7-17 on page 11 and lines 5-9 on page 15). A servlet in typical embodiments is an object, an aggregate of data elements and member methods, containing member methods for administration of digital content from sources, transcoding, selecting, and communicating the selected, transcoded digital content to client devices (described for example at lines 13-18 on page 9 and lines 11-18 on page 10).

In typical embodiments, member methods in a servlet function to receive hyperlinks to remote director instructions in the form of URLs wherein the URLs identify specific member methods, either in the servlet or in other related class objects (described for example at lines 19-22 on page 11 and lines 5-9 on page 15). The remote director instructions in such embodiments include both a URL and a member method or computer

program name. The specific member methods so identified comprise computer programs each of which is fashioned to carry out a particular task involved in transcoding, selecting, and communicating to client devices digital content from sources (described for example at line 19 on page 14 through line 3 on page 15). Carrying out a remote director instruction includes executing the computer program or member method identified by the URL of the remote director instruction (described for example at line 19 on page 14 through line 3 on page 15 and line 18 on page 16 through line 18 on page 17).

In some embodiments, the member methods identified by URLs of remote director instructions are executed as separate computational processes, so-called heavyweight processes each execution of which involved a full context switch at the operating system level. In many embodiments of the present invention, however, the member methods identified by URLs of remote director instructions are executed as lightweight threads of execution, sharing memory segments with other threads and not requiring full context switches for execution. Many embodiments implement servlets in Java at least partly because Java as a computer language has particular support for threaded program execution in the form of Java's thread-level URL dispatch routines. In typical embodiments of the present invention, the member methods identified by URLs of remote director instructions are implemented as Java thread-level URL dispatch routines (described for example at lines 1-2 on page 15). In typical embodiments, a remote director instruction comprises an instruction to select for transcoding and streaming digital content from a specific source (described for example at lines 2-3 on page 15).

Embodiments of inventive methods typically include also registering a user for a service, the service identified by a service identification code, the service comprising at least one digital content stream (described for example at references 208, 226, 234, 236, and 237 in Figure 2 and lines 3-14 on page 18); logging in the user for the service, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier (described for example at references 206, 218, 226, 228, 230, and 232 in Figure 2, references 232, 228, 242, 244, 246, 234, 236, 238, and 240 in Figure 2a, and line 16 on page 18 through line

16 on page 19); and assigning a tier value in dependence upon the device type and the service identification code (described for example at lines 18-25 on page 19); wherein the selections are dependent upon the tier (described for example at lines 19-20 on page 19); wherein transcoding further comprises transcoding in dependence upon the tier (described for example at references 102, 103, 218, 220, 224, 225, 410, and 414 in Figure 3 and lines 1-10 on page 23); and wherein communicating to at least one of the client devices the output stream further comprises communicating the output stream to the network address (described for example at lines 15-24 on page 20). In typical embodiments, registering a user includes creating a service registration record comprising service registration attributes comprising user id, service id and service subscription level and assigning a tier value further comprises assigning a tier value in dependence upon the service subscription level (described for example at reference numbers 208, 226, 234, 236, and 237 in Figure 2 and lines 18-25 on page 19).

Typical embodiments include registering a user for an event, the event in typical embodiments identified by an event identification code, the event comprising at least one digital content stream, at least one source, a start date and a start time (described for example at references 210, 226, 238, 240, 264, and 266 in Figure 2 and reference 238 in Figure 2e and lines 10-17 on page 21); logging in the user for the event, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier (described for example at references 206, 226, 228, 230, and 232 of Figure 2 and lines 1-4 on page 22); and assigning a tier value in dependence upon the device type and the event identification code (described for example at reference 240 in Figure 2 and lines 6-9 on page 22); wherein the selections are dependent upon the tier (described for example at lines 7-14 on page 24); wherein transcoding further comprises transcoding in dependence upon the tier (described for example at lines 7-14 on page 24); and wherein communicating to at least one of the client devices the output stream further comprises communicating the output stream to the network address (described for example at lines 7-14 on page 24). In typical embodiments, registering a user includes creating an event registration record comprising event registration attributes comprising user id, event id, event subscription

level, start date, and start time and assigning a tier value further comprises assigning a tier value in dependence upon the event subscription level (described for example at lines 6-9 on page 22).

All such references to the specification identify descriptions and discussions that are part of the detailed descriptions of exemplary embodiments of the present invention in the present application. Such descriptions and discussions are not limitations of the claims in the present application. The only limitations of the claims are set forth in the claims themselves.

GROUNDS OF REJECTION

Claims 1-36 in the present application are rejected in the Final Office Action dated June 30, 2005 for obviousness-type double patenting over claims 1-22 of co-pending Application No. 09/882174, over claims 10-15 of co-pending Application No. 09/881919, over claims 1-20 of co-pending Application No. 09/881917, and over claims 1-12 of co-pending Application No. 09/882173. As explained in detail below, Applicants respectfully traverse the obviousness-type double patenting rejections of the present claims.

Independent claims 1, 13, and 25 stand rejected under 35 U.S.C § 103(a) as unpatentable over a first reference entitled <u>Application Server Solution Guide, Enterprise Edition: Getting Started</u>, Nusbaum, et al., May 2000, pages 1-45, 416-434 (hereafter 'Nusbaum'), and in view of a second reference entitled <u>Java Media Framework API Guide</u>, JMP 2.0 FCS, November 19, 1999, Sun Microsystems, pages 1-66, 109-135, 173-178 (hereafter 'Sun'). As explained in detail below, Applicants respectfully traverse the rejections of the present claims under 35 USC § 103(a).

ARGUMENT

REJECTIONS FOR CLAIMS 1-36 BASED ON OBVIOUSNESS-TYPE DOUBLE PATENTING ARE IMPROPER

All claims in the present application are rejected in the Final Office Action for obviousness-type double patenting over claims 1-22 of co-pending Application No. 09/882174, over claims 10-15 of co-pending Application No. 09/881919, over claims 1-20 of co-pending Application No. 09/881917, and over claims 1-12 of co-pending Application No. 09/882173.

The law governing double patenting is that the analysis employed in an obviousness-type double patenting determination parallels the guidelines for a 35 U.S.C. § 103(a) rejection. The factual inquiries set forth in *Graham v. John Deere Co.*, 383 U.S. 1, 148 USPQ 459 (1966) are applied for establishing a background for determining obviousness under 35 U.S.C. § 103 and are employed when making an obviousness-type double patenting rejection. *Manual of Patent Examining Procedure* § 804 IIB1. The *Graham* factual inquiries require the Examiner to:

- determine the scope and content of the art as described in each co-pending application;
- determine the differences between the scope and content of the art as described in each co-pending application and the claims at issue;
- determine the level of ordinary skill in the pertinent art; and
- evaluate any objective indicia of nonobviousness.

Co-pending Application No. 09/882174

The Final Office Action rejects claims 1-36 for obviousness-type double patenting as being unpatentable over 1-22 of co-pending Application No. 09/882174. The Final Office Action at page 8 states:

> … they are not patentably distinct from each other because the limitations of the independent claims 1, 13, 26 are similar to claim 1 of co-pending Application No. 09/882174. The limitations, "remote direction of streaming digital content from a content server to a client devices using remote director" is equivalent to the use of content information, transcoding gateway for providing director instructions to stream digital content, and the use of email containing digital content. The limitations of dependent claims 2-12, 14-23, 26-36, are similar to claims 2-22 of co-pending Application No. 09/882174.

As described above, the Final Office Action must apply the *Graham* factors to establish the required background for a double patenting rejection. Although the Final Office Action does not even mention the *Graham* factors, the Final Office Action at pages 8 and 9 states:

> The co-pending application handles transcoding information using the network device. The current application also handles transcoding information using the network device. The claimed subject matter of the co-pending application does not mention about the transcoding being done using remote director instruction. However, the concept of using remote director instructions is well known in the art. For example, Nusbaum discloses usage of remote director instructions (e.g., servlet aliases, servlet URLs, sections 1.1 and 1.2, pages 1 and 2). The remote director instructions would help provide instructions to perform the transcoding from a remote device.

Applicants take the assertion in the Final Office Action that co-pending Application No. 09/882174 "handles transcoding information using the network device" as a determination under *Graham* of the scope and content of the art as described in co-pending application No. 09/882174. In response, Applicants note that the Final Office Action mischaracterizes co-pending application No. 09/882174 by asserting that the co-pending application transcodes information using a network device. Co-pending application No. 09/882174 claims a transcoding gateway for "transcoding the digital object into a digital file having a digital format and a file name...." The scope and content of the art as described in co-pending application No. 09/882174 therefore includes a transcoding gateway that transcodes a digital object into a digital file having a digital format and a file name. The scope and content of the art as described in co-pending application No. 09/882174 does not include any network device that transcodes information as asserted in the Final Office Action. Because the Final Office Action does not properly determine the scope and content of the art as described in co-pending application No. 09/882174, the Final Office Action does not establish the necessary background for determining obviousness. Without establishing the necessary background for determining obviousness, the Final Office Action cannot support an obviousness-type double patenting rejection, and the rejections of claims 1-36 should be withdrawn.

Applicants take the assertion in the Final Office Action that "[t]he current application also handles transcoding information using the network device" and that "[t]he claimed subject matter of the co-pending application does not mention about the transcoding being done using remote director instruction" as a determination under *Graham* of the differences between the scope and content of the art as described in co-pending application No. 09/882174 and the claims at issue. In response, Applicants note that the Final Office Action mischaracterizes the present application by asserting that the present application transcodes information using a network device. The present application claims a content server for "transcoding, in dependence upon the remote director's instructions, the digital content from sources into digital content having streaming format...." The scope and content of the art as described in the present application therefore includes a content server that transcodes, in dependence upon the remote

director's instructions, the digital content from sources into digital content having streaming format, not any network device that transcodes information as asserted in the Final Office Action. Because the Final Office Action does not properly determine the scope and content of the art as described in the present application, the Final Office Action cannot determine the differences between the scope and content of the art as described in co-pending application No. 09/882174 and the claims at issue. The Final Office Action therefore does not establish the necessary background for determining obviousness. Without establishing the necessary background for determining obviousness, the Final Office Action cannot support an obviousness-type double patenting rejection, and the rejections of claims 1-36 should be withdrawn.

Applicants further assume that the Final Office Action at pages 8 and 9 attempts to determine under *Graham* the level of ordinary skill in the pertinent art by stating:

> However, the concept of using remote director instructions is well known in the art. For example, Nusbaum discloses usage of remote director instructions (e.g., servlet aliases, servlet URLs, sections 1.1 and 1.2, pages 1 and 2). The remote director instructions would help provide instructions to perform the transcoding from a remote device.

The Final Office Action cites sections 1.1 and 1.2 of Nusbaum in an attempt to determine that remote director instructions are within the level of ordinary skill in the pertinent art. In response, Applicants note that Nusbaum is not 'pertinent art' available for determining under *Graham* the level of ordinary skill in the pertinent art because Nusbaum is not in the field of the Applicants' endeavor or reasonably pertinent to the particular problem with which the Applicants were concerned. In fact, Nusbaum cannot be a reference against the claims of the present application because Nusbaum does actually represent nonanalogous art within the meaning of *In Re Horn*, *Clay*, and *Oeitker*. *In re Horn*, 203 USPQ 969 (CCPA 1979), *In re Clay*, 966 F.2d 656, 23 USPQ2d 1058 (Fed. Cir. 1992), *In re Oeticker*, 977 F.2d 1443, 24 USPQ2d 1443 (Fed. Cir. 1992). The field of the inventors' effort in this case is streaming digital content under remote direction. The

present application claims, among other things, receiving digital content from the sources, the digital content having a multiplicity of digital formats, receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream, and transcoding, in dependence upon the remote director's instructions, the digital content from sources into digital content having streaming format. The field of Nusbaum is dynamic web pages for the World Wide Web – which has nothing to do with the technical field of the present application and is not reasonably pertinent to the particular problem with which the Applicants were concerned. Because Nusbaum is neither within the field of the inventor's endeavor in this case nor reasonably pertinent to the particular problem with which the Applicants were concerned, Nusbaum is not pertinent art available for determining the level of ordinary skill in the pertinent art under *Graham*. The Final Office Action therefore cannot establish the necessary background for determining obviousness and cannot support an obviousness-type double patenting rejection. The rejections of claims 1-36 should be withdrawn.

Even if Nusbaum was pertinent art, which Nusbaum is not, sections 1.1 and 1.2 of Nusbaum do not disclose usage of remote director instructions as claimed in the present application. What section 1.1 of Nusbaum actually discloses is a general use of the term 'application' in the context of IBM's WebSphere® product that provides integration and application infrastructure. What section 1.2 of Nusbaum actually discloses is a general description of Java servlets. A Java servlet is a software module that extends request/response oriented servers, such as Java-enabled web servers. Sections 1.1 and 1.2 of Nusbaum have nothing to do with remote director instructions. Nusbaum's general use of an application in the context of WebSphere® and Nusbaum's general description of Java servlets does not disclose remote director instructions as claimed in the present application. Because sections 1.1 and 1.2 of Nusbaum do not disclose remote director instructions as claimed in the present application, the Final Office Action does not properly establish the level of ordinary skill in the pertinent art. The Final Office Action therefore does not establish the necessary background for determining obviousness and

cannot support an obviousness-type double patenting rejection. The rejections of claims 1-36 should be withdrawn.

Co-Pending Application No. 09/881919

The Final Office Action rejects claims 1-36 for obviousness-type double patenting as being unpatentable over claims 10-15 of co-pending Application No. 09/881919. The Final Office Action at page 9 states:

> ... they are not patentably distinct from each other because the limitations of the independent claims 1, 13, 26 are similar to claim 10 of co-pending Application No. 09/881919. The limitations, "remote direction of streaming digital content from a content server to a client devices using remote director" is equivalent to the use of a content server through which digital content is transcoded into streams of multimedia data, the streams communicated via network to client devices, use of the digital content for streaming, use of remote director instructions comprising hyperlinked URSs invoked through a network-capable device. The limitations of dependent claims 2-12, 14-23, 26-36, are similar to claims 11-15 of co-pending Application No. 09/881919.

As described above, the Final Office Action must apply the *Graham* factors to establish the required background for a double patenting rejection. Although the Final Office Action does not even mention the *Graham* factors, the Final Office Action does recite the same assertions made regarding co-pending Application No. 09/882174 above by stating:

> The co-pending application handles transcoding information using the network device. The current application also handles transcoding information using the network device. The claimed subject matter of the co-pending application does not mention about the transcoding being done using remote director instruction. However, the concept of using remote director instructions is well known in the art. For example, Nusbaum

> discloses usage of remote director instructions (e.g., servlet aliases, servlet URLs, sections 1.1 and 1.2, pages 1 and 2). The remote director instructions would help provide instructions to perform the transcoding from a remote device.

Applicants take such assertions as an attempt to apply the *Graham* factors regarding this co-pending Application No. 09/881919. In response, Applicants note that the Final Office Action does not establish the necessary background for determining obviousness for the reasons discussed above with regard to co-pending Application No. 09/882174. The Final Office Action therefore cannot support an obviousness-type double patenting rejection, and the rejections of claims 1-36 should be withdrawn.

<u>Co-Pending Application No. 09/881917</u>

The Final Office Action rejects claims 1-36 for obviousness-type double patenting as being unpatentable over claims 1-20 of co-pending Application No. 09/881917. The Final Office Action at page 10 states:

> … they are not patentably distinct from each other because the limitations of the independent claims 1, 13, 26 are similar to claim 1 of co-pending Application No. 09/881917. The limitations, "remote direction of streaming digital content from a content server to a client devices using remote director" is equivalent to the use of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices, use of network of digital computers comprising a content server. The limitations of dependent claims 2-12, 14-23, 26-36, are similar to claims 2-20 of co-pending Application No. 09/881917.

As described above, the Final Office Action must apply the *Graham* factors to establish the required background for a double patenting rejection. Although the Final Office Action does not even mention the *Graham* factors, the Final Office Action does recite the same assertions made regarding co-pending Application No. 09/882174 above by stating:

> The co-pending application handles transcoding information using the network device. The current application also handles transcoding information using the network device. The claimed subject matter of the co-pending application does not mention about the transcoding being done using remote director instruction. However, the concept of using remote director instructions is well known in the art. For example, Nusbaum discloses usage of remote director instructions (e.g., servlet aliases, servlet URLs, sections 1.1 and 1.2, pages 1 and 2). The remote director instructions would help provide instructions to perform the transcoding from a remote device.

Applicants take such assertions as an attempt to apply the *Graham* factors regarding this co-pending Application No. 09/881917. In response, Applicants note that the Final Office Action does not establish the necessary background for determining obviousness for the reasons discussed above with regard to co-pending Application No. 09/882174. The Final Office Action therefore cannot support an obviousness-type double patenting rejection, and the rejections of claims 1-36 should be withdrawn.

Co-Pending Application No. 09/882173

The Final Office Action rejects claims 1-36 for obviousness-type double patenting as being unpatentable over claims 1-12 of co-pending Application No. 09/882173. The Final Office Action at page 11 states:

> … they are not patentably distinct from each other because the limitations of the independent claims 1, 13, 26 are similar to claim 1 of co-pending Application No. 09/882173. The limitations, "remote direction of streaming digital content from a content server to a client devices using remote director" is equivalent to the use of remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices upon a network of digital

> computer comprising a content server receiving digital content from the sources and the digital content having a multiplicity of digital formats. The limitations of dependent claims 2-12, 14-23, 26-36, are similar to claims 2-12 of co-pending Application No. 09/882173.

As described above, the Final Office Action must apply the *Graham* factors to establish the required background for a double patenting rejection. Although the Final Office Action does not even mention the *Graham* factors, the Final Office Action does recite the same assertions made regarding co-pending Application No. 09/882174 above by stating:

> The co-pending application handles transcoding information using the network device. The current application also handles transcoding information using the network device. The claimed subject matter of the co-pending application does not mention about the transcoding being done using remote director instruction. However, the concept of using remote director instructions is well known in the art. For example, Nusbaum discloses usage of remote director instructions (e.g., servlet aliases, servlet URLs, sections 1.1 and 1.2, pages 1 and 2). The remote director instructions would help provide instructions to perform the transcoding from a remote device.

Applicants take such assertions as an attempt to apply the *Graham* factors regarding this co-pending Application No. 09/882173. In response, Applicants note that the Final Office Action does not establish the necessary background for determining obviousness for the reasons discussed above with regard to co-pending Application No. 09/882174. The Final Office Action therefore cannot support an obviousness-type double patenting rejection, and the rejections of claims 1-36 should be withdrawn.

Conclusion

The Final Office Action does not establish the necessary background for determining obviousness required by an obviousness-type double patenting rejection of claims 1-36 in the present application over claims 1-22 of co-pending Application No. 09/882174, over claims 10-15 of co-pending Application No. 09/881919, over claims 1-20 of co-pending Application No. 09/881917, and over claims 1-12 of co-pending Application No. 09/882173. Based on the reasoning provided in the Final Office Action, no person of ordinary skill in the art would conclude that claims 1-36 in the present case are obvious in view of claims 1-22 of co-pending Application No. 09/882174, over claims 10-15 of co-pending Application No. 09/881919, over claims 1-20 of co-pending Application No. 09/881917, and over claims 1-12 of co-pending Application No. 09/882173. Applicants therefore respectfully traverse the rejections of claims 1-36 and request reconsideration of claims 1-36 in light of the present remarks.

REJECTIONS FOR CLAIMS 1-36 BASED ON OBVIOUSNESS UNDER 35 U.S.C § 103(a) ARE IMPROPER

Claims 1-36 are in the case. Independent claims 1, 13, and 25 stand rejected for obviousness under 35 U.S.C § 103(a) as unpatentable over a first reference entitled Application Server Solution Guide, Enterprise Edition: Getting Started, Nusbaum, et al., May 2000, pages 1-45, 416-434 (hereafter 'Nusbaum'), and in view of a second reference entitled Java Media Framework API Guide, JMP 2.0 FCS, November 19, 1999, Sun Microsystems, pages 1-66, 109-135, 173-178 (hereafter 'Sun'). To establish a prima facie case of obviousness, three basic criteria must be met. *Manual of Patent Examining Procedure* §2142. The first element of a prima facie case of obviousness under 35 U.S.C. § 103 is that Nusbaum and Sun must teach or suggest all of Applicants' claim limitations. *In re Royka*, 490 F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974). The second element of a prima facie case of obviousness under 35 U.S.C. § 103 is that there must be a suggestion or motivation to combine Nusbaum and Sun. *In re Vaeck*, 947 F.2d 488, 493, 20 USPQ2d 1438, 1442 (Fed. Cir. 1991). The third element of a prima facie

case of obviousness under 35 U.S.C. § 103 is that there must be a reasonable expectation of success in the proposed combination of Nusbaum and Sun. *In re Merck & Co., Inc.*, 800 F.2d 1091, 1097, 231 USPQ 375, 379 (Fed. Cir. 1986). As demonstrated below, the combination of Nusbaum and Sun does not establish a prima facie case of obviousness. The rejection of independent claims 1, 13, and 25 should therefore be withdrawn and the case should be allowed.

Nusbaum and Sun Do Not Teach
Each and Every Element of the Claim

To establish a prima facie case of obviousness, the proposed combination of Nusbaum and Sun must disclose all of applicants' claim limitations. *In re Royka*, 490F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974). Independent claim 1 of the present application claims:

> 1. A method of remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices the method implemented upon a network of digital computers, at least one of the digital computers comprising a content server upon which the steps of the method are implemented in computer memory and at least one computer processor, the method comprising the steps of:
>
> receiving digital content from the sources, the digital content having a multiplicity of digital formats;
>
> receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream;

carrying out the remote director instructions, wherein carrying out the remote director instructions further comprises:

selecting, in dependence upon the remote director's instructions, digital content for inclusion in an output stream;

transcoding, in dependence upon the remote director's instructions, the digital content from sources into digital content having streaming format;

including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format; and

communicating, in dependence upon the remote director's instructions, to at least one of the client devices the output stream.

In rejecting claims 1, 13, and 25, the Final Office Action at page 13 states that Nusbaum teaches "remote direction (e.g., figure 5, page 13)…." What Figure 5 on page 13 of Nusbaum actually depicts is a general "EJB environment and interaction with other components," where 'EJB' stands for Enterprise JavaBeans™. An Enterprise JavaBeans™ is a server-side object that conforms to the Enterprise JavaBeans™ Specification. The EJB specification describes the server-side component architecture of the Java 2 Enterprise Edition ('J2EE') platform that provides a framework for components to "plug in" to a server and extend that server's functionality. A server-side object that conforms to the Enterprise JavaBeans™ Specification is not remote direction of streaming digital content as claimed in the present application. Nusbaum's EJB environment and EJB interaction with other components therefore does not teach remote direction of streaming digital content as claimed in the present application. The Final Office Action does not produce references that teach each and every element of independent claims 1, 13, and 25 and the rejections should be withdrawn.

In rejecting claims 1, 13 and 25, the Final Office Action at pages 13 and 14 also states that Nusbaum teaches:

> receiving, from a remote director, and storing in computer memory, remote director instructions (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417)
>
> . . .
>
> the remote director instructions including instructions for selections of digital content for inclusion in an output streaming (e.g., section 1.2.4, page 6, sections 2.1.1.1, pages 31 and 32, section 8.1.8, page 417)
>
> . . .
>
> carrying out the remote director instructions (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417), wherein carryout out the remote director instructions further comprises:
>
> selecting, in dependence upon the remote director's instructions, digital content for inclusion in an output stream (e.g., section 1.2.4, page 6)
>
> . . .
>
> in dependence upon the remote director's instructions handling the digital content from sources (e.g. section 2.1.1.1, pages 31 and 32)
>
> . . .
>
> including in an output streaming, in dependence upon the remote director's instructions, to at least one of the client devices the output stream (e.g. section 1.2.4, page 6)
>
> . . .
>
> communicating, in dependence upon the remote director's instructions, to at least one of the client devices the output stream (e.g., section 1.2.4, page 6).

That is, the Final Office Action relies on Nusbaum at section 1.2.4 on page 6, section 2.1.1.1 on pages 31 and 32, and section 8.1.8 on page 417 to teach the following claim elements and limitations:

- receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream

- carrying out the remote director instructions, wherein carrying out the remote director instructions further comprises

- selecting, in dependence upon the remote director's instructions, digital content for inclusion in an output stream

- transcoding, in dependence upon the remote director's instructions, the digital content from sources into digital content having streaming format

- including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format

- communicating, in dependence upon the remote director's instructions, to at least one of the client devices the output stream

What section 1.2.4 of Nusbaum actually teaches is an application programming interface ('API') for a JavaServlet™. A servlet is a small Java computer program running in a server environment that allows a software developer to add dynamic content to a web server. Such Java technology for generating dynamic web pages is known as Java Server Pages, or 'JSP.' Section 2.1.1.1 of Nusbaum actually teaches an EJB server architecture that includes an "EJB server runtime," "EJB containers," and "Enterprise Java beans" that together provide services such as a "deployment tool," "naming services," and "security services." Section 8.1.8 of Nusbaum actually teaches a "DMT interface" for

"connect[ing] to one or more directory servers…," where 'DMT' stands for Directory Management Tool. Nusbaum's details of the JavaServlet™ API, EJB server architecture, and directory management tool interface have nothing whatsoever to do with the elements and limitations of the claims of the present invention for which the Final Office Action cites the Nusbaum references. In fact, Nusbaum never once mentions remote direction of streaming digital content, remote directors, or remote director instructions as claimed in the present application. The fact that Nusbaum makes some general references to network communications or that Sun makes general references to streaming media is completely insufficient to anticipate or suggest claim elements in the present application. Nusbaum's servlet API details, EJB server architecture, and directory management tool interface therefore does not teach the elements and limitations of the claims of the present application as cited in the Final Office Action. Because The Final Office Action does not produce references that teach each and every element of independent claims 1, 13, and 25, the rejections should be withdrawn.

In rejecting claims 1, 13, and 25, the Final Office Action at page 14 states that Sun teaches:

> Sun teaches streaming digital content and transcoding (e.g., transcoding the video contents, page 33) into digital content having streaming format objects and the digital content having a multiplicity of digital formats (e.g., streaming media as per remote instruction, page 4, MPEG, JPEG, etc., video formatted content, page 6).

That is, the Final Office Action takes the position that pages 4, 6, and 33 of Sun teach "digital content having a multiplicity of digital formats…" and "transcoding…digital content from sources into digital content having streaming format…" as claimed in the present application. What page 4 of Sun actually teaches is definitions to some common terms used in streaming media including 'content type,' 'media stream,' 'multiplex,' 'track,' and so on. Page 6 of Sun merely discloses a chart of common video and audio formats. What page 33 of Sun actually teaches is a definition of transcoding as a

"process of converting each track of media data from one input format to another." Sun's definition of transcoding and of some common streaming media terms along with Sun's chart of common video and audio formats does not teach "digital content having a multiplicity of digital formats..." and "transcoding...digital content from sources into digital content having streaming format..." as claimed in the present application. The Final Office Action does not produce references that teach each and every element of independent claims 1, 13, and 25 and the rejections should be withdrawn.

The Cited References Set Forth No Suggestion
To Combine Nusbaum and Sun

To establish a prima facie case of obviousness, there must be a suggestion or motivation to modify or combine Nusbaum and Sun. *In re Vaeck*, 947 F.2d 488, 493, 20 USPQ2d 1438, 1442 (Fed. Cir. 1991). "The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination." In re Mills, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). The Examiner has not pointed to any disclosure in Nusbaum or Sun suggesting the desirability of the combination. Moreover, there is no possibility whatsoever that the Examiner could ever point to any disclosure in Nusbaum or Sun suggesting the desirability of the combination. Nusbaum in fact makes no mention whatsoever of remote direction of streaming digital content, remote directors, or remote director instructions, and therefore could not possibly suggest the desirability of the combination. In addition, no such suggestion occurs in Sun. Absent such a showing of desirability to combine Nusbaum and Sun, the Examiner has impermissibly used "hindsight" occasioned by Applicants' own teaching to reject the claims. *In re Surko*, 11 F.3d 887, 42 U.S.P.Q.2d 1476 (Fed. Cir. 1997); *In re Vaeck*, 947 F.2d 488, 20 U.S.P.Q.2d 1438 (Fed. Cir. 1991); *In re Gorman*, 933 F.2d 982, 986, 18 U.S.P.Q.2d 1885, 1888 (Fed. Cir. 1991); *In re Bond*, 910 F.2d 831, 15 U.S.P.Q.2d 1566 (Fed. Cir. 1990); *In re Laskowski*, 871 F.2d 115, 117, 10 U.S.P.Q.2d 1397, 1398 (Fed. Cir. 1989). The proposed combination of Nusbaum and Sun therefore cannot possibly establish a prima facie case of obviousness. The objection should be withdrawn, and the case should be allowed.

There Is No Reasonable Expectation Of Success In The Proposed Combination Of Nusbaum And Sun

To establish a prima facie case of obviousness, there must be a reasonable expectation of success in the proposed combination of Nusbaum and Sun. *In re Merck & Co., Inc.,* 800 F.2d 1091, 1097, 231 USPQ 375, 379 (Fed. Cir. 1986). The Examiner has not pointed to any disclosure in Nusbaum or Sun suggesting any expectation of success in such a combination. In fact, there can be no reasonable expectation of success in the proposed combination of Nusbaum and Sun because Nusbaum and Sun cannot be combined to provide remote direction of streaming digital content as claimed in the present application. The Final Office Action bases this rejection on portions of Nusbaum that includes Figure 5 on page 13, section 1.2.4 on page 6, section 2.1.1.1 on pages 31 and 32, and section 8.1.8 on page 417. As explained above, these references to Nusbaum teach an EJB execution environment and a kind of dynamic web page technology known as Java Server Pages or 'JSP.' The Final Office Action also bases this rejection on portions of Sun that includes page 4, page 6, and page 33, which as explained above, merely provide some definitions regarding streaming media as the terms are used in the Java Media Framework. As described in Nusbaum, dynamic web page technology is methods and systems for building server pages on the fly. Dynamic web pages generally, and JSPs in particular, is not streaming media and does not combine with streaming media to provide remote direction of streaming digital content as claimed in the present application. The proposed combination of Nusbaum and Sun therefore cannot support a prima facie case of obviousness. The rejection should be withdrawn, and the case should be allowed.

Nusbaum Teaches Away From the Claims of the Present Application

Turning now to the substance of Nusbaum, Nusbaum actually teaches away from the current application. Teaching away from the claims is a *per se* demonstration of lack of

prima facie obviousness. *In re Dow Chemical Co.,* 837 F.2d 469, 5 U.S.P.Q.2d 1529 (Fed. Cir. 1988); *In re Fine,* 837 F.2d 1071, 5 U.S.P.Q.2d 1596 (Fed. Cir. 1988); *In re Neilson,* 816 F.2d 1567, 2 U.S.P.Q.2d 1525 (Fed. Cir. 1987). Nusbaum discloses dynamic web page technology with no mention of remote directors or remote director instructions. Without even mentioning remote directors or remote director instructions, there could be no impulse on the part of developers of dynamic web page technology to incorporate remote directors or remote director instructions into dynamic web page technology. By effecting dynamic web page technology alone, with no hint or suggestion that remote directors or remote director instructions might even exist, Nusbaum teaches away from the combination with Sun proposed in the Final Office Action. Because Nusbaum teaches away from the Applicants' claims, the proposed modification of Nusbaum with Sun cannot support a prima facie case of obviousness. The rejection of Applicants' claims should be withdrawn, and the case should be allowed.

<u>Nusbaum Cannot be a Reference Against the Claims of the Present Application Because Nusbaum Represents Nonanalogous Art</u>

Nusbaum cannot be a reference against the claims of the present application because Nusbaum represents nonanalogous art within the meaning of *In Re Horn*, *In re Clay*, and *In re Oeitker*. *In re Horn*, 203 USPQ 969 (CCPA 1979), *In re Clay*, 966 F.2d 656, 23 USPQ2d 1058 (Fed. Cir. 1992), *In re Oeticker*, 977 F.2d 1443, 24 USPQ2d 1443 (Fed. Cir. 1992). The field of the inventors' effort in this case is remote direction of streaming digital content. The present application claims, among other things, receiving digital content, receiving remote director instructions, and carrying out the remote director instructions. The field of Nusbaum is dynamic web pages for the World Wide Web – which clearly has nothing to do with the technical field of the present application. Nusbaum therefore is not within the field of the inventor's endeavor in this case.

Because Nusbaum is not within the field of the inventor's endeavor in this case, there can be no basis for believing that Nusbaum as a reference would have been considered by one skilled in the particular art working on the relevant problem to which this invention

pertains. That is, there would be no reason for an inventor concerned with remote direction of streaming digital content to search for art regarding dynamic generation of web pages. The two simply have nothing to do with one another. Nusbaum as a reference therefore is not reasonably pertinent to the particular problem with which the inventors were involved in the present case and is not available as a reference against the present application. Applicants respectfully propose that for this reason alone the rejection of the present claims should be withdrawn, and the claims should be allowed.

Conclusion

All claims in the present case stand rejected under 35 U.S.C § 103(a). Independent claims 1, 13, and 25 stand rejected under 35 U.S.C § 103(a) over Nusbaum in view of Sun. The combination of Nusbaum and Sun fails to establish a prima face case of obviousness. The applicants have demonstrated that it is incorrect to reject the independent claims 1, 13, and 25 under 35 U.S.C § 103(a). The dependent claims of the present application include each and every element and limitation of the independent claims from which they depend. Because the Final Office Action cites only the combination of Nusbaum and Sun to teach or suggest each and every element of the independent claims and because all the independent claims stand, Applicants respectfully propose that all the dependent claims in the present case stand. The rejection of all the claims 1-36 should therefore be withdrawn, and the claims should be allowed. Applicants respectfully traverse the rejection of claims 1-36 and request reconsideration of claims 1-36 in light of the present remarks.

APPPLICANTS' RESPONSE TO EXAMINER'S RESPONSE TO APPLICANTS' RESPONSE TO THE FIRST OFFICE ACTION DATED SEPTEMBER 24, 2004

The Final Office Action responds to Applicants' Response to First Office Action of September 24, 2004 ("First Office Action") with the following arguments. First, Nusbaum and Sun set forth a suggestion or motivation to combine Nusbaum and Sun. Second, there is a reasonable expectation of success in the combination of Nusbaum and

Sun. Third, Nusbaum and Sun teach or suggest each and every element and limitation of the Applicants' claims. Finally, Nusbaum is analogous art because it is in the field of Applicants' endeavor or reasonably pertinent to the particular problem with which the Applicants were concerned. Applicants respectfully traverse each of the responses in the Final Office Action to Applicants' Response to the First Office Action and respond below in detail to the new arguments set forth in the Final Office Action.

<u>The Cited References Set Forth No Suggestion Or Motivation To Combine Nusbaum And Sun</u>

The Final Office Action at pages 2 and 3 argues that Nusbaum and Sun are properly combined for an obviousness rejection under 35 U.S.C. § 103 on the grounds that:

> Nusbaum teaches a method, a system, and a computer program product to implement remote direction (e.g., figure 5, page 13) of handling information from a multiplicity of sources of digital information to a multiplicity of client devices (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417) the method implemented upon a network of digital computers (e.g., figure 5, page 13, at least one of the digital computers comprising a content server upon which the steps of the method are implemented (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417) in computer memory and at least one computer processor (e.g., server containing web content, page 13) the method comprising the steps of: receiving digital content from the sources (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417) receiving, from a remote director, and storing in computer memory, remote director instructions (e.g. 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417), the remote director instructions including instructions for selections of digital content for inclusion in an output streaming (e.g., section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417); carrying out the remote director instructions

(e.g. section 1.2.4, page 6, section 2.1.1.1, pages 31 and 32, section 8.1.8, page 417). Sun teaches well known concept of streaming digital content and transcoding (e.g., transcoding the video contents, page 33) into digital content having streaming format objects and the digital content having a multiplicity of digital formats (e.g., streaming media, page 4, MPEG, JPEG, etc., video formatted content, page 6).

That is, the Final Office Action responds to Applicants' argument that there is no suggestion or motivation to combine Nusbaum and Sun by stating that Nusbaum and Sun teach various elements and limitations of the independent claims. As explained in detail above, the cited references in fact do not teach remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices as claimed in the present application. Nusbaum generally teaches a kind of dynamic web page technology known as Java Server Pages. What Nusbaum teaches specifically at Figure 5 is an Enterprise JavaBean™ environment and its interaction with other network components. Section 1.2.4 of Nusbaum specifically teaches an application programming interface ('API') for a JavaServlet™. Section 2.1.1.1 of Nusbaum specifically teaches an Enterprise JavaBean™ server architecture, while section 8.1.8 of Nusbaum specifically teaches a Directory Management Tool interface. Sun generally teaches the Java Media API, an application programming interface for deliver of time-based media. What Sun teaches specifically at pages 4, 6, and 33 is a general description of streaming media and content types, common audio formats, and a general description of media player operations. Not only do the cited portions of the references fail to disclose or suggest elements of the present claims, even if they did so, the Final Office Action cannot arbitrarily pick and choose with massive hindsight elements of Applicants' claims from Java Server Pages and the Java Media API and use them as a basis to conclude the present claims invalid for obviousness. For these reasons, Applicants continue to assert that there is no suggestion or motivation to combine Nusbaum and Sun. The proposed combination of Nusbaum and Sun therefore cannot support a prima facie case of obviousness. The rejections to claims 1-36 should be withdrawn, and the case should be allowed.

The Final Office Action at page 2 citing *In re Keller*, 642 F.2d 413, 425, 208 USPQ 871, 881 (CCPA 1981) and *In re Young*, 927 F.2d 588, 591 18 USPQ2d 1089, 1091 (Fed. Cir. 1991), also argues that:

> the test for obviousness is not whether the features of a secondary reference may be bodily incorporated into the structure of a primary reference. It is also not that the claimed invention must be expressly suggested in any one or all of the references. Rather, the test is what the combined teachings of the references would have suggested to those of skill in the art.

In this way, the Final Office Action implicitly argues that there is no need for the Examiner to demonstrate that the references provide motivation or suggestion to combine or that there is any reasonable expectation of success in combining the references so long as elements of the present claims are disclosed in the references.

As the Commissioner is well aware, however, such is not the law. The mere fact that references can be combined or modified does not render the resultant combination obvious unless the prior art also suggests the desirability of the combination. *In re Mills*, 916 F.2d 680, 16 USPQ2d 1430 (Fed. Cir. 1990). In fact, the requirement of a prima facie case of obviousness places a burden on the examiner to provide some suggestion of the desirability of doing what the inventor has done. "To support the conclusion that the claimed invention is directed to obvious subject matter, either the references must expressly or impliedly suggest the claimed invention or the examiner must present a convincing line of reasoning as to why the artisan would have found the claimed invention to have been obvious in light of the teachings of the references." *Ex parte Clapp*, 227 USPQ 972, 973 (Bd. Pat. App. & Inter. 1985). When the motivation to combine the teachings of the references is not immediately apparent, it is the duty of the examiner to explain why the combination of the teachings is proper. *Ex parte Skinner*, 2 USPQ2d 1788 (Bd. Pat. App. & Inter. 1986); MPEP § 2142.

The Final Office Action merely continues the practice begun in the First Office Action of pointing to elements of method and system in its cited references and stating that they are the same things claimed in the present patent application. The Final Office Action makes no substantive attempt whatsoever to present a prima facie case of obviousness by pointing to express or implicit suggestion to combine in the references themselves or by explaining or providing any basis for concluding that persons of skill in the art would be moved to combine the references. For these reasons also, Applicants continue to assert that the cited references do not contain a suggestion or motivation to combine. The proposed combination of Nusbaum and Sun therefore cannot support a prima facie case of obviousness. The rejections to claims 1-36 should be withdrawn, and the case should be allowed.

There Is No Reasonable Expectation Of Success In The Combination Of Nusbaum And Sun

In response to Applicants' Response to the First Office Action, the Final Office Action at pages 3 and 4 argues that there is a reasonable expectation of success in the combination of Nusbaum and Sun. The Final Office Action does not, however, provide any new basis for this assertion other than stating that Nusbaum and Sun teach various elements and limitations of the independent claims. As explained in detail above, the cited references in fact do not teach remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices as claimed in the present application. Nusbaum generally teaches a kind of dynamic web page technology known as Java Server Pages. Sun generally teaches the Java Media API, an application programming interface for delivery of time-based media. The dynamic web page technology of Nusbaum for building adhoc server pages is not related to the Java API for time-based media of Sun. In fact, the technology for building dynamic web pages is entirely different from a Java API for time-based media. JSPs generate documents viewable in a web browser, while the API for time-based media provide a Java class architecture to software developers. Combining Nusbaum's Java Server Pages and Sun's

Java Media API does not provide remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices as claimed in the present application. For these reasons, Applicants continue to assert that there is therefore no reasonable expectation of success in the proposed combination of Nusbaum and Sun. The proposed combination of Nusbaum and Sun therefore cannot support a prima facie case of obviousness. The rejections to claims 1-36 should be withdrawn, and the case should be allowed.

Nusbaum And Sun Do Not Teach Each and Every Element of the Claim

The Final Office Action at page 5 again argues that Nusbaum and Sun teach each and every element of independent claims 1, 13, and 25 by reciting portions of Nusbaum and Sun mentioned above in this Brief. To establish a prima facie case of obviousness, the proposed combination of Nusbaum and Sun must teach all of Applicants' claim limitations. *In re Royka*, 490F.2d 981, 985, 180 USPQ 580, 583 (CCPA 1974). As discussed in detail above, the combination of Nusbaum and Sun does not teach each and every element of the independent claims in the present case. The Final Office Action therefore cannot establish a prima facie case for obviousness of claims 1, 23, and 45 under 35 U.S.C. § 103. The dependent claims of the present application include each and every element and limitation of the independent claims from which they depend. Because the Final Office Action cites only the combination of Nusbaum and Sun to teach or suggest each and every element of the independent claims and because all the independent claims stand, Applicants respectfully propose that all the dependent claims in the present case stand. The proposed combination of Nusbaum and Sun therefore cannot support a prima facie case of obviousness. The rejections to claims 1-36 should be withdrawn, and the case should be allowed.

Nusbaum Is Not Analogous Art Because It Is Neither In The Field Of Applicants' Endeavor Nor Reasonably Pertinent To The Particular Problem With Which The Applicants Were Concerned

In response to Applicants' First Office Action, the Final Office Action at pages 6 and 7 argues that Nusbaum is analogous art available for rejecting the claims of the present application. The Final Office Action asserts that Nusbaum is in the field of Applicants' endeavor or reasonably pertinent to the particular problem with which the Applicants were concerned. In support for this assertion, the Final Office Action at page 6 and 7 only states that Nusbaum and Sun teach various elements and limitations of the independent claims. As explained in detail above, the cited references in fact do not teach remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices as claimed in the present application. Nusbaum generally teaches a kind of dynamic web page technology known as Java Server Pages. Sun generally teaches the Java Media API, an application programming interface for delivery of time-based media. Without more, Nusbaum's Java Server Pages does not place Nusbaum in the field of Applicants' endeavor or make Nusbaum reasonably pertinent to the particular problem with which the Applicants were concerned, such concern being, remote direction of streaming digital content as claimed in the present application.

In addition, Applicants respectfully propose that Nusbaum cannot be a reference against the claims of the present application because Nusbaum actually represents nonanalogous art within the meaning of *In Re Horn*, *Clay*, and *Oeitker*. *In re Horn*, 203 USPQ 969 (CCPA 1979), *In re Clay*, 966 F.2d 656, 23 USPQ2d 1058 (Fed. Cir. 1992), *In re Oeticker*, 977 F.2d 1443, 24 USPQ2d 1443 (Fed. Cir. 1992). The field of the inventors' effort in this case is remote direction of streaming digital content. The present application claims, among other things, receiving digital content, receiving remote director instructions, and carrying out the remote director instructions. The field of Nusbaum is dynamic web pages for the World Wide Web – which clearly has nothing to

do with the technical field of the present application. Nusbaum therefore is not within the field of the inventor's endeavor in this case.

Because Nusbaum is not within the field of the inventor's endeavor in this case, there can be no basis for believing that Nusbaum as a reference would have been considered by one skilled in the particular art working on the relevant problem to which this invention pertains. That is, there would be no reason for an inventor concerned with remote direction of streaming digital content to search for art regarding dynamic generation of web pages. The two simply have nothing to do with one another. Nusbaum as a reference therefore is neither within the field of the Applicants' endeavor nor reasonably pertinent to the particular problem with which the inventors were involved in the present case. Nusbaum therefore is not available as a reference against the present application. Applicants respectfully propose that for this reason alone the rejection of the present claims 1-36 should be withdrawn, and the claims should be allowed.

<u>Conclusion</u>

Applicants traverse each of the arguments in the Final Office Action responding to Applicants' Response to First Office Action. Applicants demonstrate that the cited references in the Final Office Action set forth neither a suggestion nor motivation to combine Nusbaum and Sun. Furthermore, there is no reasonable expectation of success in the combination of Nusbaum and Sun. In addition, Nusbaum and Sun do not teach or suggest each and every element and limitation of the Applicants' claims. Applicants also demonstrate that Nusbaum represents nonanalogous art. Applicants' therefore respectfully traverse each rejection individually of claims 1-36.

In view of the forgoing arguments, reversal on all grounds of rejection is requested.

The Commissioner is hereby authorized to charge or credit Deposit Account No. 09-0447 for any fees required or overpaid.

Respectfully submitted,

Date: November 30, 2005

By: John Biggers

John Biggers
Reg. No. 44,537
Biggers & Ohanian, LLP
P.O. Box 1469
Austin, Texas 78767-1469
Tel. (512) 472-9881
Fax (512) 472-9887
ATTORNEY FOR APPELLANTS

APPENDIX OF CLAIMS
ON APPEAL IN PATENT APPLICATION OF
WILLIAM K. BODIN, *ET AL.*, SERIAL NO. 09/881,915

CLAIMS

What is claimed is:

1. A method of remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices the method implemented upon a network of digital computers, at least one of the digital computers comprising a content server upon which the steps of the method are implemented in computer memory and at least one computer processor, the method comprising the steps of:

receiving digital content from the sources, the digital content having a multiplicity of digital formats;

receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream;

carrying out the remote director instructions, wherein carrying out the remote director instructions further comprises:

selecting, in dependence upon the remote director's instructions, digital content for inclusion in an output stream;

transcoding, in dependence upon the remote director's instructions, the digital content from sources into digital content having streaming format;

including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format;

communicating, in dependence upon the remote director's instructions, to at least one of the client devices the output stream.

2. The method of claim 1, wherein the client devices comprise client device attributes, said transcoding further comprising transcoding in dependence upon the client device attributes.

3. The method of claim 2 wherein client device attributes include device type, screen size, frame rate, and audio status.

4. The method of claim 1 wherein the remote director comprises a personal computer coupled through a network to the content server, the method further comprising:

sending from the remote director to the content server remote director instructions, further comprising invoking through URLs displayed on a terminal of the remote director member methods in servlets installed on the content server.

5. The method of claim 4 wherein the invoking through URLs further comprises invoking through each URL a single member method in a servlet.

6. The method of claim 5 wherein the single member method is programmed to

carry out a single remote director instruction.

7. The method of claim 5 wherein the single member method is implemented as a Java thread-level URL dispatch routine.

8. The method of claim 4 wherein the remote director instruction comprises an instruction to select for transcoding and streaming digital content from a specific source.

9. The method of claim 1 further comprising the steps of:

registering a user for a service, the service identified by a service identification code, the service comprising at least one output stream;

logging in the user for the service, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

assigning a tier value in dependence upon the device type and the service identification code;

wherein the selections are dependent upon the tier;

wherein transcoding further comprises transcoding in dependence upon the tier; and

wherein communicating to at least one of the client devices the output stream further comprises communicating the output stream to the network address.

10. The method of claim 6 wherein:

registering a user further comprises creating a service registration record comprising service registration attributes comprising user id, service id and service subscription level; and

assigning a tier value further comprises assigning a tier value in dependence upon the service subscription level.

11. The method of claim 1 further comprising the steps of:

registering a user for an event, the event identified by an event identification code, the event comprising at least one output stream, at least one source, a start date and a start time;

logging in the user for the event, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

assigning a tier value in dependence upon the device type and the event identification code;

wherein the selections are dependent upon the tier;

wherein transcoding further comprises transcoding in dependence upon the tier; and

wherein communicating to at least one of the client devices the output stream further comprises communicating the output stream to the network address.

12. The method of claim 5 wherein:

registering a user further comprises creating an event registration record comprising event registration attributes comprising user id, event id, event subscription level, start date, and start time; and

assigning a tier value further comprises assigning a tier value in dependence upon the event subscription level.

13. A system for remote direction of streaming digital content from a multiplicity of sources of digital information to a multiplicity of client devices the system implemented upon a network of digital computers, at least one of the digital computers comprising a content server upon which the system is implemented in computer memory and upon at least one computer processor, the system comprising:

means for receiving digital content from the sources, the digital content having a multiplicity of digital formats;

means for receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream;

means for transcoding the digital content from sources into digital content having streaming format;

means for including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format;

means for communicating to at least one of the client devices the output stream.

14. The system of claim 13, wherein the client devices comprise client device attributes, said means for transcoding further comprising means for transcoding in

dependence upon the client device attributes.

15. The system of claim 14wherein client device attributes include device type, screen size, frame rate, and audio availability.

16. The system of claim 13 wherein the remote director comprises a personal computer coupled through a network to the content server, the system further comprising:

means for sending from the remote director to the content server remote director instructions, further comprising means for invoking through URLs displayed on a terminal of the remote director member methods in servlets installed on the content server.

17. The system of claim 16 wherein the means for invoking through URLs further comprises means for invoking through each URL a single member method in a servlet.

18. The system of claim 17 wherein the single member method is programmed to carry out a single remote director instruction.

19. The system of claim 17 wherein the single member method is implemented as a thread-level Java URL dispatch routine.

20. The system of claim 16 wherein the remote director instruction comprises an instruction to select for transcoding and streaming digital content from a specific source.

21. The system of claim 13 further comprising:

means for registering a user for a service, the service identified by a service identification code, the service comprising at least one output stream;

means for logging in the user for the service, said means for logging in the user further comprising means for assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

means for assigning a tier value in dependence upon the device type and the service identification code;

wherein the selections are dependent upon the tier;

wherein means for transcoding further comprises means for transcoding in dependence upon the tier; and

wherein means for communicating to at least one of the client devices the output stream further comprises means for communicating the output stream to the network address.

22. The system of claim 18wherein:

means for registering a user further comprises means for creating a service registration record comprising service registration attributes comprising user id, service id and service subscription level; and

means for assigning a tier value further comprises means for assigning a tier value in dependence upon the service subscription level.

23. The system of claim 13further comprising:

means for registering a user for an event, the event identified by an event identification code, the event comprising at least one output stream, at least one source, a start date and a start time;

means for logging in the user for the event, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

means for assigning a tier value in dependence upon the device type and the event identification code;

wherein the selections are dependent upon the tier;

wherein means for transcoding further comprises means for transcoding in dependence upon the tier; and

wherein means for communicating to at least one of the client devices the output stream further comprises means for communicating the output stream to the network address.

24. The system of claim 17 wherein:

means for registering a user further comprises means for creating an event registration record comprising event registration attributes comprising user id, event id, event subscription level, start date, and start time; and

means for assigning a tier value further comprises means for assigning a tier value in dependence upon the event subscription level.

25. A computer program product for remote direction of streaming digital content

from a multiplicity of sources of digital information to a multiplicity of client devices the system implemented upon a network of digital computers, at least one of the digital computers comprising a content server upon which the system is implemented in computer memory and upon at least one computer processor, the computer program product comprising:

a recording medium;

means, recorded on the recording medium, for receiving digital content from the sources, the digital content having a multiplicity of digital formats;

means, recorded on the recording medium, for receiving, from a remote director, and storing in computer memory, remote director instructions, the remote director instructions including instructions for selections of digital content for inclusion in an output stream;

means, recorded on the recording medium, for transcoding the digital content from sources into digital content having streaming format;

means, recorded on the recording medium, for including in an output stream, in dependence upon the remote director's instructions, digital content having streaming format;

means, recorded on the recording medium, for communicating to at least one of the client devices the output stream.

26. The computer program product of claim 25, wherein the client devices comprise client device attributes, said means for transcoding further comprising means for transcoding in dependence upon the client device attributes.

27. The computer program product of claim 26 wherein client device attributes

include device type, screen size, frame rate, and audio availability.

28. The computer program product of claim 25 wherein the remote director comprises a personal computer coupled through a network to the content server, the system further comprising:

means, recorded on the recording medium, for sending from the remote director to the content server remote director instructions, further comprising means, recorded on the recording medium, for invoking through URLs displayed on a terminal of the remote director member methods in servlets installed on the content server.

29. The computer program product of claim 28 wherein the means for invoking through URLs further comprises means for invoking through each URL a single member method in a servlet.

30. The computer program product of claim 29 wherein the single member method is programmed to carry out a single remote director instruction.

31. The computer program product of claim 29 wherein the single member method is implemented as a thread-level Java URL dispatch routine.

32. The computer program product of claim 28 wherein the remote director instruction comprises an instruction to select for transcoding and streaming digital content from a specific source.

33. The computer program product of claim 25further comprising:

means, recorded on the recording medium, for registering a user for a service, the service identified by a service identification code, the service comprising at least one output stream;

means, recorded on the recording medium, for logging in the user for the service, said means for logging in the user further comprising means for assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

means, recorded on the recording medium, for assigning a tier value in dependence upon the device type and the service identification code;

wherein the selections are dependent upon the tier;

wherein means for transcoding further comprises means for transcoding in dependence upon the tier; and

wherein means for communicating to at least one of the client devices the output stream further comprises means for communicating the output stream to the network address.

34. The computer program product of claim 30 wherein:

means for registering a user further comprises means for creating a service registration record comprising service registration attributes comprising user id, service id and service subscription level; and

means for assigning a tier value further comprises means for assigning a tier value in dependence upon the service subscription level.

35. The computer program product of claim 25further comprising:

means, recorded on the recording medium, for registering a user for an event, the event identified by an event identification code, the event comprising at least one output stream, at least one source, a start date and a start time;

means, recorded on the recording medium, for logging in the user for the event, logging in the user further comprising assigning values to user login attributes, the user login attributes comprising user identification, device type, network address, and a tier;

means, recorded on the recording medium, for assigning a tier value in dependence upon the device type and the event identification code;

wherein the selections are dependent upon the tier;

wherein means for transcoding further comprises means for transcoding in dependence upon the tier; and

wherein means for communicating to at least one of the client devices the output stream further comprises means for communicating the output stream to the network address.

36. The computer program product of claim 29 wherein:

means for registering a user further comprises means for creating an event registration record comprising event registration attributes comprising user id, event id, event subscription level, start date, and start time; and

means for assigning a tier value further comprises means for assigning a tier value in dependence upon the event subscription level.

A Bytecode Translator for Distributed Execution of "Legacy" Java Software
Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba and Kozo Itano
In Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001), June, 2001.
LNCS 2072, ECOOP 2001 - Object Oriented Programming, page 236-255, Springer-Verlag, 2001.
Get Copy Get Slide

サーバのアクセス制限を安全に変更するための機構
光来 健一, 千葉 滋
情報処理学会 論文誌, 42巻6号, 1492-1502頁, 2001年6月.
(in Japanese)
Get Copy

A Secure Access Control Mechanism against Internet Crackers
Kenichi Kourai and Shigeru Chiba
In Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21), pages 743-746, April 2001.
Get Copy Get Poster

通信機構に合わせた最適化を行う並列化コンパイラ
横田大輔, 千葉滋, 板野肯三
情報処理学会 論文誌, 42巻4号, 860-867頁, 2001年4月.
(in Japanese)
Get Copy

2000

クラスオブジェクトを用いた Java 言語用マクロ処理系
立堀 道昭, 千葉 滋, 板野 肯三
情報処理学会 論文誌, 41巻8号, pp.2327-2338, 2000年8月.
(in Japanese)
Get Copy

OpenJava: A Class-based Macro System for Java
Michiaki Tatsubori, Shigeru Chiba, Marc-Orivier Killijian and Kozo Itano
In Reflection and Software Engineering, W.Cazzola, R.J. Stroud, and F.Tisato, Eds., LNCS 1826, Springer-Verlag, pages 119--135, July 2000.
Get Copy

Load-time Structural Reflection in Java

Publications

- Refereed
- Workshop/Poster/Tech Report
- Thesis

Refereed Papers

2004

感性を考慮したジョブスケジューリング
栗田亮・千葉滋・光来健一
情報処理学会論文誌：コンピューティングシステム、vol.45, no. SIG3(ACS 5), Mar. 2004, pp.29-37. (in Japanese)
Get Copy

Remote Pointcut — A Language Construct for Distributed AOP
Muga Nishizawa and Shigeru Chiba
In Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04), pp.7-16. Lancaster, UK. March 22-26, 2004.
Get Copy Get Slide

Josh: An Open AspectJ-like Language
Shigeru Chiba and Kiyoshi Nakagawa
In Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04), pp.102-112. Lancaster, UK. March 22-26, 2004.
Get Copy

2003

Secure and Manageable Virtual Private Networks for End-users
Kenichi Kourai, Toshio Hirotsu, Koji Sato, Osamu Akashi, Kensuke Fukuda, Toshiharu Sugawara, and Shigeru Chiba
In Proceedings of the 28th Annual IEEE Conference on Local Computer Networks (LCN 2003), pp.385-394, 2003.
Get Copy Get Slide

効率的な Java Dynamic AOP システムを実現する Just-in-Time Weaver

佐藤 芳樹, 千葉 滋,
情報処理学会論文誌:プログラミング, vol. 44, no. SIG 13 (PRO18), 15-24 項, 2003年10月.
(in Japanese)
Get Copy Get Slide

A Selective, Just-in-Time Aspect Weaver
Yoshiki Sato and Shigeru Chiba and Tatsubori Michiaki
Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.189-208, Springer-Verlag, 2003.
Get Copy Get Slide

An Easy-to-Use Toolkit for Efficient Java Bytecode Translators
Shigeru Chiba and Muga Nishizawa
Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.364-376, Springer-Verlag, 2003.
Get Copy

2002

A New Optimization Technique for the Inspector-Executor Method
Daisuke Yokota, Shigeru Chiba and Kozo Itano
Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, 2002.
Proceedings of the 14th IASTED International Conference, page 711-716.
Get Copy Get Slide

Addistant：アスペクト指向の分散プログラミング支援ツール
立堀 道昭, 千葉 滋, 板野肯三
情報処理学会 研究会論文誌 Programming, 43巻3(PRO14)号, 17-25頁, 2002年3月.
(in Japanese)
Get Copy

2001

Javaバイトコード変換による構造リフレクションの実現
千葉 滋、立堀 道昭
情報処理学会 論文誌, 42巻11号, 2752-2760頁, 2001年11月.
(in Japanese)
Get Copy

OpenJava: A Class-based Macro System for Java

Michiaki Tatsubori[1], Shigeru Chiba[2], Marc-Olivier Killijian[3], and Kozo Itano[2]

[1] Doctral Program in Engineering, University of Tsukuba,
Tennohdai 1-1-1, Tsukuba, Ibaraki, Japan,
mt@is.tsukuba.ac.jp
[2] Department of Information Science and Electronics, University of Tsukuba
[3] Laboratory for Analysis and Architecture of Systems,
Centre National de la Recherche Scientifique

Abstract. This paper presents OpenJava, which is a macro system that we have developed for Java. With traditional macro systems designed for non object-oriented languages, it is difficult to write a number of macros typical in object-oriented programming since they require the ability to access a logical structure of programs. One of the drawbacks of traditional macro systems is that abstract syntax trees are used for representing source programs. This paper first points out this problem and then shows how OpenJava addresses this problem. A key idea of OpenJava is to use metaobjects, which was originally developed for reflective computing, for representing source programs.

1 Introduction

Reflection is a technique for changing the program behavior according to another program. From software engineering viewpoint, reflection is a tool for separation of concerns and thus it can be used for letting programmers write a program with higher-level abstraction and with good modularity. For example, a number of reflective systems provide metaobjects for intercepting object behavior, that is, method invocations and field accesses. Those metaobjects can be used for *weaving* several programs seprately written from distinct aspects, such as an application algorihtm, distribution, resource allocation, and user interface, into a single executable program.

However, previous reflective systems do not satisfy all the requirements in software engineering. Although the abstraction provided by the metaobjects for intercepting object behavior is easy to understand and use, they can be used for implementing only limited kinds of separation of concerns. Moreover, this type of reflection often involves runtime penalties. Reflective systems should enable more fine-grained program weaving and perform as much reflective computation as possible at compile time for avoiding runtime penalities.

On the other hand, a typical tool for manipulating a program at compile time has been a macro system. It performs textual substitution so that a particualr aspect of a program is separated from the rest of that program. For example, the C/C++ macro system allows to separate the definition of a constant value

from the rest of a program, in which that constant value is used in a number of distinct lines. The Lisp macro system provides programable macros, which enables more powerful program manipulation than the C/C++ one. Also, since macro expansion is done at compile time, the use of macros does not imply any runtime penalties. However, the abstraction provided by traditional macro systems is not sophisticated; since macros can deal with only textual representation of a program, program manipulation depending on the semantic contexts of the program cannot be implemented with macros.

This paper proposes a macro system integrating good features of the reflective approach, in other words, a compile-time reflective system for not only behavioral reflection but also structural reflection. A key idea of our macro system, called *OpenJava*, is that macros (meta programs) deal with class metaobjects representing logical entities of a program instead of a sequence of tokens or abstract syntax trees (ASTs). Since the class metaobjects abstract both textual and semantic aspects of a program, macros in OpenJava can implement more fine-grained program weaving than in previous reflective systems. They can also access semantic contexts if they are needed for macro expansion. This paper presents that OpenJava can be used to implement macros for helping complex programmings with a few design patterns.

In the rest of this paper, section 2 presents a problem of ordinary macro systems and section 3 discusses the design and implementation of OpenJava, which addresses this problem. We compare OpenJava with related work in section 4. Finally, section 5 concludes this paper.

2 Problems with Ordinary Macros

Macro systems have been typical language-extension mechanisms. With C/C++'s `#define` macro system, programmers can specify a symbol or a function call to be replaced with another expression, although this replacement is simple token-based substitution. In Common Lisp, programmers can write more powerful macros. However, even such powerful macros do not cover all requirements of OO languages programming.

2.1 Programmable Macros

Macros in Common Lisp are programmable macros. They specify how to replace an original expression in Common Lisp itself. A macro function receives an AST (abstract syntax tree) and substitutes it for the original expression. Since this macro system is powerful, the object system of Common Lisp (CLOS) is implemented with this macro system.

Programmable macros have been developed for languages with more complex syntax like C. MS^2[19] is one of those macro systems for C. Macro functions are written in an extended C language providing special data structure representing ASTs. The users of MS^2 can define a new syntax and how it is expanded into

a regular C syntax. The parameter that a macro function receives is an AST of the code processed by that macro function.

One of the essential issue in designing a programmable macro system is a data structure representing an original source program. Another essential issue is how to specify where to apply each macro in a source program. For the former, most systems employed ASTs. For the latter, several mechanisms were proposed.

In Common Lisp and MS^2, a macro is applied to expressions or statements beginning with the trigger word specified by the macro. For example, if the trigger word is `unless`, all expressions beginning with `unless` are expanded by that macro. In this way, they cannot use macros without the trigger words. For instance, it is impossible to selectively apply a macro to only + expressions for adding string objects.

Some macro systems provide fine-grained control of where to apply a macro. In A^* [14], a macro is applied to expressions or statements matching a pattern specified in the BNF. In EPP[9], macros are applied to a specified syntax elements like `if` statements or + expressions. There's no need to put any trigger word in front of these statements or expressions.

2.2 Representation of Object-Oriented Programs

Although most of macro systems have been using ASTs for representing a source program, ASTs are not the best representation for all macros: some macros typical in OO programming require a different kind of representation. ASTs are purely textual representation and independent of logical or contextual information of the program. For example, if an AST represents a binary expression, the AST tells us what the operator and the operands are but it never tells us the types of the operands. Therefore, writing a macro is not possible with ASTs if the macro expansion depends on logical and contextual information of that binary expression.

There is a great demand for the macros depending on logical and contextual information in OO programming. For example, some of design patterns[6] require relatively complicated programming. They often require programmers to repeatedly write similar code.[1] To help this programming, several researchers have proposed to extend a language to provide new language constructs specialized for particular patterns [1, 7]. Those constructs should be implemented with macros although they have been implemented so far by a custom preprocessor. This is because macros implementing those constructs depend on the logical and contextual information of programs and thus they are not implementable on top of the traditional AST-based macro systems.

Suppose that we write a macro for helping programming with the OBSERVER[6] pattern, which is for describing one-to-many dependency among objects. This pattern is found in the Java standard library although it is called the event-and-listener model. For example, a Java program displays a menu bar must define a listener object notified of menu-select events. The listener object is an instance of a class MyMenuListener implementing interface MenuListener:

```
class MyMenuListener implements MenuListener {
   void menuSelected(MenuEvent e) { .. }
   void menuDeselected(MenuEvent e) { return; }
   void menuCanceled(MenuEvent e) { return; }
}
```

This class must declare all the methods for event handling even though some events, such as the menu cancel event, are simply ignored.

We write a macro for automating declaration of methods for handling ignored events. If this macro is used, the definition of MyMenuListener should be rewritten into:

```
class MyMenuListener follows ObserverPattern
    implements MenuListener
{
    void menuSelected(MenuEvent e) { .. }
}
```

The `follows` clauses specifies that our macro ObserverPattern is applied to this class definition. The declarations of `menuDeselected()` and `menuCanceled()` are automated. This macro first inspects which methods declared in the interface MenuListener are not implemented in the class MyMenuListener. Then it inserts the declarations of these methods in the class MyMenuListener.

Writing this macro is difficult with traditional AST-based macro systems since it depends on the logical information of the definition of the class MyMenuListener. If a class definition is given as a large AST, the macro program must interpret the AST and recognize methods declared in MenuListener and MyMenuListener. The macro program must also construct ASTs representing the inserted methods and modify the original large AST to include these ASTs. Manipulating a large AST is another difficult task. To reduce these difficulties, macro systems should provide logical and contextual information of programs for macro programs. There are only a few macro systems providing the logical information. For example, XL[15] is one of those systems although it is for a functional language but not for an OO language.

3 OpenJava

OpenJava is our advanced macro system for Java. In OpenJava, macro programs can access the data structures representing a logical structure of the programs. We call these data structure class metaobjects. This section presents the design of OpenJava.

3.1 Macro Programming in OpenJava

OpenJava produces an object representing a logical structure of class definition for each class in the source code. This object is called a class metaobject. A class

metaobject also manages macro expansion related to the class it represents. Programmers customize the definition of the class metaobjects for describing macro expansion. We call the class for the class metaobject *metaclass*. In OpenJava, the metaprogram of a macro is described as a metaclass. Macro expansion by OpenJava is divided into two: the first one is macro expansion of class declarations (callee-side), and the second one is that of expressions accessing classes (caller-side).

Applying Macros Fig. 1 shows a sample using a macro in OpenJava. By adding a clause `instantiates M` in just after the class name in a class declaration, the programmer can specify that the class metaobject for the class is an instance of the metaclass M. In this sample program, the class metaobject for MyMenuListener is an instance of ObserverClass. This metaobject controls macro expansion involved with MyMenuListener. The declaration of ObserverClass is described in regular Java as shown in Fig. 2.

```
class MyMenuListener
   instantiates ObserverClass
   extends MyObject
   implements MenuListener
{ .... }
```

Fig. 1. Application of a macro in OpenJava

```
class ObserverClass
   extends OJClass
{
   void translateDefinition() { ... }
   ....
}
```

Fig. 2. A macro in OpenJava

Every metaclass must inherit from the metaclass OJClass, which is a built-in class of OpenJava. The `translateDefinition()` in Fig. 2 is a method inherited from OJClass, which is invoked by the system to make macro expansion. If an `instantiates` clause in a class declaration is found, OpenJava creates an instance of the metaclass indicated by that `instantiates` clause, and assigns this instance to the class metaobject representing that declared class. Then OpenJava invokes `translateDefinition()` on the created class metaobject for macro expansion on the class declaration later.

Since the translateDefinition() declared in OJClass does not perform any translation, a subclass of OJClass must override this method for the desired macro expansion. For example, translateDefinition() can add new member methods to the class by calling other member methods in OJClass. Modifications are reflected on the source program at the final stage of the macro processing.

Describing a Metaprogram The method translateDefinition() implementing the macro for the OBSERVER pattern in section 2.2 is shown in Fig. 3. This metaprogram first obtains all the member methods (including inherited ones) defined in the class by invoking getMethods() on the class metaobject. Then, if a member method declared in interfaces is not implemented in the class, it generates a new member method doing nothing and adds it to the class by invoking addMethod() on the class metaobject.

```
void translateDefinition() {
   OJMethod[] m = this.getMethods(this);
   for (int i = 0; i < m.length; ++i) {
      OJModifier modif = m[i].getModifiers();
      if (modif.isAbstract()) {
         OJMethod n = new OJMethod(this,
            m[i].getModifiers().removeAbstract(),
            m[i].getReturnType(), m[i].getName(),
            m[i].getParameterTypes(),
            m[i].getExceptionTypes(),
            makeStatementList("return;"));
         this.addMethod(n);
      }
   }
}
```

Fig. 3. translateDefinition() in ObserverClass

As a class is represented by a class metaobjects, a member method is also represented by a method metaobjects. In OpenJava, classes, member methods, member fields, and constructors are represented by instances of the class OJClass, OJMethod, OJField, and OJConstructor, respectively. These metaobject represent logical structures of class and member definitions. They are easy to handle, compared to directly handling large ASTs representing class declarations and collecting information scattered in these ASTs.

3.2 Class Metaobjects

As shown in section 2, a problem of ordinary macro systems is that their primary data structure is ASTs (abstract syntax trees) but they are far from logical sturctures of programs in OO languages. In OO languages like Java, class definitions play an important role as a logical structure of programs. Therefore, OpenJava employs the class metaobjects model, which was originally developed for reflective computing, for representing a logical structure of a program. The

class metaobjects make it easy for meta programs to access a logical structure of program.

Hiding Syntactical Information In Java, programmers can use various syntax for describing the logically same thing. These syntactical differences are absorbed by the metaobjects. For instance, there are two notations for declaring a String array member field:

```
String[] a;
String b[];
```

Both a and b are String array fields. It would be awkward to write a metaprogram if the syntactical differences of the two member fields had to be considerd. Thus OJField provides only two member methods `getType()` and `setType()` for handling the type of a member field. `getType()` on the OJField metaobjects representing a and b returns a class metaobject representing the array type of the class String.

Additionally, some elements in the grammer represent the same element in a logical structure of the language. If one of these element is edited, the others are also edited. For instance, the member method `setName()` in OJClass for modifying the name of the class changes not only the class name after the `class` keyword in the class declaration but also changes the name of the constructors.

Logically Structured Class Representation Simple ASTs, even arranged and abstracted well, cannot properly represent a logical structure of a class definition. The data structure must be carefully designed to corresponded not only to the grammer of the language but also to the logical constructs of the language like classes and member methods. Especially, it makes it easy to handle the logical information of program including association between names and types.

For instance, the member method `getMethods()` in OJClass returns all the member methods defined in the class which are not only the methods immediately declared in the class but also the inherited methods. The class metaobjects contain type information so that the definition of the super class can be accessible.

3.3 Class Metaobjects in Details

The root class for class metaobjects is OJClass. The member methods of OJClass for obtaining information about a class are shown in Tab. 1 and Tab. 2. They cover all the attributes of the class. In OpenJava, all the types, including array types and primitive types like `int`, have corresponding class metaobjects. Using the member methods shown in Tab. 1, metaprogramms can inspect whether a given type is an ordinary class or not.

Tab. 3 gives methods for modifying the definition of the class. Metaprograms can override `translateDefinition()` in OJClass so that it calls these methods

for executing desired modifications. For instance, the example shown in Fig. 3 adds newly generated member methods to the class with `addMethod()`.

Table 1. Member Methods in OJClass for Non-Class Types

`boolean isInterface()`	Tests if this represents an interface type.
`boolean isArray()`	Tests if this represents an array type.
`boolean isPrimitive()`	Tests if this represents an premitive type.
`OJClass getComponentType()`	Returns a class metaobject for the type of array components.

Metaobjects Obtained through Class Metaobjects The method `getSuperclass()` in OJClass, which is used to obtain the superclass of the class, returns a class metaobject instead of the class name (as a string). As the result, metaprogram can use the returned class metaobject to directly obtain information about the superclass. OpenJava automatically generates class metaobjects on demand, even for classes declared in another source file or for classes available only in the form of bytecode, that is, classes whose source code is not available.

The returned value of the member method `getModifiers()` in Tab. 2 is an instance of the class OJModifier. This class represents a set of class modifiers such as `public`, `abstract` or `final`. Metaprogramms do not have to care about the order of class modifiers because OJModifier hides such useless information.

The class OJMethod, which is the return type of `getDeclaredMethods()` in OJClass, represents a logical structure of a method. Thus, similarly to the class OJClass, this class has member methods for examining or modifying the attributes of the method. Some basic member methods in OJMethod are shown in Tab. 4. Any type information obtained from these methods is also represented by a class metaobject. For instance, `getReturnType()` returns a class metaobject as the return type of the method. This feature of OJMethod is also found in OJField and OJConstructor, which respectively represent a member field and a constructor.

The class StatementList, which is the return type of the member method `getBody()` in the class OJMethod, represents the statements in a method body. An instance of StatementList consists of objects representing either expressions or statements. StatementList objects are AST-like data structures although they contain type information. This is because we thought that the logical structure of statements and expressions in Java can be well represented with ASTs.

Table 2. Member Methods in OJClass for introspection (1)

`String getPackageName()`
Returns the package name this class belongs to.

`String getSimpleName()`
Returns the unqualified name of this class.

`OJModifier getModifiers()`
Returns the modifiers for this class.

`OJClass getSuperclass()`
Returns the superclass declared explicitly or implicitly.

`OJClass[] getDeclaredInterfaces()`
Returns all the declared superinterfaces.

`StatementList getInitializer()`
Returns all the static initializer statements.

`OJField[] getDeclaredFields()`
Returns all the declared fields.

`OJMethod[] getDeclaredMethods()`
Returns all the declared methods.

`OJConctructor[] getDeclaredConstructors()`
Returns all the constructors declared explicitly or implicitly.

`OJClass[] getDeclaredClasses()`
Returns all the member classes (inner classes).

`OJClass getDeclaringClass()`
Returns the class declaring this class (outer class).

Table 3. Member Methods in OJClass for modifying the class

`String setSimplename(String name)`
Sets the unqualified name of this class.

`OJModifier setModifiers(OJModifier modifs)`
Sets the class modifiers.

`OJClass setSuperclass(OJClass clazz)`
Sets the superclass.

`OJClass[] setInterfaces(OJClass[] faces)`
Sets the superinterfaces to be declared.

`OJField removeField(OJField field)`
Removes the given field from this class declaration.

`OJMethod removeMethod(OJMethod method)`
Removes the given method from this class declaration.

`OJConstructor removeConstructor(OJConstructor constr)`
Removes the given constructor from this class declaration.

`OJField addField(OJField field)`
Adds the given field to this class declaration.

`OJMethod addMethod(OJMethod method)`
Adds the given method to this class declaration.

`OJConstructor addConstructor(OJConstructor constr)`
Adds the given constructor to this class declaration.

Table 4. Basic Methods in OJMethod

`String getName()`
Returns the name of this method.

`OJModifier getModifiers()`
Returns the modifiers for this method.

`OJClass getReturnType()`
Returns the return type.

`OJClass[] getParameterTypes()`
Returns the parameter types in declaration order.

`OJClass[] getExceptionTypes()`
Returns the types of the exceptions declared to be thrown.

`String[] getParameterVariables()`
Returns the parameter variable names in declaration order.

`StatementList getBody()`
Returns the statements of the method body.

`String setName(String name)`
Sets the name of this method.

`OJModifier setModifiers(OJModifier modifs)`
Sets the method modifiers.

`OJClass setReturnType()`
Sets the return type.

`OJClass[] setParameterTypes()`
Sets the parameter types in declaration order.

`OJClass[] setExceptionTypes()`
Sets the types of the exceptions declared to be thrown.

`String[] setParameterVariables()`
Sets the parameter variable names in declaration order.

`StatementList setBody()`
Sets the statements of the method body.

Logical Structure of a Class Tab. 5 shows the member methods in OJClass handling a logical structure of a class. Using these methods, metaprograms can obtain information considering class inheritance and member hiding. Although these member methods can be implemented by combining the member methods in Tab.2, they are provided for convenience. We think that providing these methods is significant from the viewpoint that class metaobjects represent a logical structure of a program.

Table 5. Member Methods in OJClass for introspection (2)

`OJClass[] getInterfaces()` Returns all the interfaces implemented by this class or the all the superinterfaces of this interface.
`boolean isAssignableFrom(OJClass clazz)` Determines if this class/interface is either the same as, or is a superclass or superinterface of, the given class/interface.
`OJMethod[] getMethods(OJClass situation)` Returns all the class available from the given situation, including those declared and those inherited from superclasses/superinterfaces.
`OJMethod getMethod(String name, OJClass[] types, OJClass situation)` Returns the specified method available from the given situation.
`OJMethod getInvokedMethod(String name, OJClass[] types, OJClass situation)` Returns the method, of the given name, invoked by the given arguments types, and available from the given situation.

In considering the class inheritance mechanism, the member methods defined in a given class are not only the member methods described in that class declaration but also the inherited ones. Thus, method metaobjects obtained by invoking `getMethods()` on a class metaobject include the methods explicitly declared in its class declaration but also the methods inherited from its superclass or superinterfaces.

Moreover, accessibility of class members is restricted in Java by member modifiers like `public`, `protected` or `private`. Thus, `getMethods()` returns only the member methods available from the class specified by the argument. For instance, if the specified class is not a subclass or in the same package, `getMethods()` returns only the member methods with `public` modifier. In Fig. 3, since the metaprogram passes `this` to `getMethods()`, it obtains all the member methods defined in that class.

3.4 Type-Driven Translation

As macro expansion in OpenJava is managed by metaobjects corressponding to each class (type), this translation is said to be type-driven. In the above example, only the member method `translateDefinition()` of OJClass is overridden to translate the class declarations of specified classes (callee-side translation).

In addition to the callee-side translation, `OJClass` provides a framework to translate the code related to the corresponding class spreaded over whole program selectively (caller-side translation). The parts related to a certain class is, for example, instance creation expressions or field access expressions.

Here, we take up an example of a macro that enables programming with the FLYWEIGHT[6] pattern to explain this mechanism. This design pattern is applied to use objects-sharing to support large numbers of fine-grained objects efficiently. An example of macro supporting uses of this pattern would need to translate an instance creation expression of a class Glyph:

```
new Glyph('c')
```

into a class method call expression:

```
GlyphFactory.createCharacter('c')
```

The class method `createCharacter()` returns an object of Glyph correspondent to the given argument if it was already generated, otherwise it creates a new object to return. This way, the program using Glyph objects automatically shares an object of Glyph representing a font for a letter c without generating several objects for the same letter. In ordinary programming using Glyph objects with the FLYWEIGHT pattern, programmers must explicitly write `createCharacter()` in their program with creations of Glyph objects. With a support of this macro, instance creations can be written in the regular `new` syntax and the pattern is used automatically.

In OpenJava, this kind of macro expansions are implemented by defining a metaclass FlyweightClass to be applied to the class Glyph. This metaclass overrides the member method `expandAllocation()` of OJClass as in Fig.4. This method receives a class instance creation expression and returns a translated expression. The system of OpenJava examines the whole source code and apply this member method to each Glyph instance creation expression to perform the macro expansion.

```
Expression expandAllocation(AllocationExpression expr, Environment env) {
   ExpressionList args = expr.getArguments();
   return new MethodCall(this, "createCharacter", args);
}
```

Fig. 4. Replacement of class instance expressions

The member method `expandAllocation()` receives an AllocationExpression object representing a class instance creation expression and an Environment object representing the enviroment of this expression. The Environment object holds name binding information such as type of variable in the scope of this expression.

OpenJava uses type-driven translation to enable the complehensive macro expansion of partial code spreaded over various places in program. In macro systems for OO programing languages, it is not only needed to translate a class

declaration simply but translating expressions using the class togather is also needed. In OpenJava, by defining a methods like `expandAllocation()`, metaprogrammers can selectively apply macro expansion to the limited expressions related to classes controlled by the metaclass. This kind of mechanism has not been seen in most of ordinary macro systems except some systems like OpenC++[3]. Tab. 6 shows the primary member methods of OJClass which can be overridden for macro expansion at caller-side.

Table 6. Member Methods for Each Place Applied the Macro-Expansion to

Member method	Place applied the macro expansion to
`translateDefinition()`	Class declaration
`expandAllocation()`	Class instance allocation expression
`expandArrayAllocation()`	Array allocation expression
`expandTypeName()`	Class name
`expandMethodCall()`	Method class expression
`expandFieldRead()`	Field-read expression
`expandFieldWrite()`	Field-write expression
`expandCastedExpression()`	Casted expression from this type
`expandCastExpression()`	Casted expression to this type

3.5 Translation Mechanism

Given a source program, the processor of OpenJava:

1. Analyzes the source program to generate a class metaobject for each class.
2. Invokes the member methods of class metaobjects to perform macro expansion.
3. Generates the regular Java source program reflecting the modification made by the class metaobjects.
4. Executes the regular Java compiler to generate the corresponding byte code.

The Order of Translations Those methods of OJClass whose name start from expand performs caller-side translation, and they affect expressions in source program declaring another class C. Such expressions may also be translated by `translateDefinition()` of the class metaobject of C as callee-side translation. Thus different class metaobjects affect the same part of source program.

In OpenJava, to resolve this ambiguousness of several macro expansion, the system always invokes `translateDefinition()` first as callee-side translation, then it apply caller-side translation to source code of class declarations which was already applied callee-side translation. Metaprogrammers can design metaprogram considering this specified order of translation. In this rule,

if `translateDefinition()` changes an instance creation expression of class X into Y's, `expandAllocation()` defined in the metaclass of X is not performed.

Moreover, the OpenJava system always performs `translateDefinition()` for superclasses first, i.e. the system performs it for subclasses after superclasses. As a class definition strongly depends on the definition of its superclass, the translation of a class often varies depending on the definition of its superclass. To settle the definition of superclasses, the system first translates the source program declaring superclasses. Additionally, there are some cases where the definition of a class D affects the result of translation of a class E. In OpenJava, from `translateDefinition()` for E, metaprogrammer can explicitly specify that `translateDefinition()` for D must be performed before.

In the case there are dependency relationships of translation among several macro expansions, consistent order of translation is specified to address this ambiguousness of translation results.

Dealing with Separate Compilation In Java, classes can be used in program only if they exist as source code or byte code (.class file). If there is no source code for a class C, the system cannot specifiy the metaclass of C, as is. Then, for instance, it cannot perform the appropriate `expandAllocation()` on instance creation expressions of C.

Therefore, OpenJava automatically preserves metalevel information such as the metaclass name for a class when it processes the callee-side translation of each class. These preservation are implemented by translating these information into a string held in a field of a special class, which is to be compiled into byte code. The system uses this byte code to obtain necessary metalevel information in another process without source code of that class. Additionally, metaprogrammers can request the system to preserve customized metalevel information of a class.

Metalevel information can be preserved as special attributes of byte code. In OpenJava, such information is used only at compile-time but not at runtime. Thus, in order to save runtime overhead, we choosed to preserve such information in separated byte code which is not to be loaded by JVM at runtime.

3.6 Syntax Extension

With OpenJava macros, a metaclass can introduce new class/member modifiers and clauses starting with the special word at some limited positions of the regular Java grammar. The newly introduced clauses are valid only in the parts related to instances of the metaclass.

In a class declaration (callee-side), the positions allowed to introduce new clauses are:

- before the block of member declarations,
- before the block of method body in each method declaration,
- after the field variable in each field declaration.

And in other class declarations (caller-side), the allowed position is:

– after the name of the class.

Thanks to the limited positions of new clauses, the system can parse source programs without conflicts of extended grammers. Thus, metaprogrammers do not have to care about conflicts between clauses.

```
class VectorStack instantiates AdapterClass
    adapts Vector in v to Stack
{
    ....
}
```

Fig. 5. An example of syntax extension in OpenJava

Fig. 5 shows an example source program using a macro, a metaclass AdapterClass, supporting programming with the ADAPTER pattern[6]. The metaclass introduces a special clause beginning with adapts to make programmers to write special description for the ADAPTER pattern in the class declaration. The adapts clause in the Fig. reffig:VectorStack VectorStack is the adapter to a class Stack for a class Vector. The information by this clause is used only when the class metaobjects representing VectorStack performs macro expansion. Thus, for other class metaobjects, semantical information added by the new clause is recognized as a regular Java source code.

```
static SyntaxRule getDeclSuffix(String keyword) {
    if (keyword.equals("adapts")) {
        return new CompositeRule(
            new TypeNameRule(),
            new PrepPhraseRule("in", new IdentifierRule()),
            new PrepPhraseRule("to", new TypeNameRule()) );
    }
    return null;
}
```

Fig. 6. A meta-program for a customized suffix

To introduce this adapts clause, metaprogrammers implement a member method `getDeclSuffix()` in the metaclass AdapterClass as shown in Fig. 6. The member method `getDeclSuffix()` is invoked by the system when needed, and returns a SyntaxRule object representing the syntax grammer begenning with the given special word. An instance of the class SyntaxRule implements a recursive descendant parser of LL(k), and analyzes a given token series to generate an appropriate AST. The system uses SyntaxRule objects obtained by invoking `getDeclSuffix()` to complete the parsing.

For metaprogrammers of such SyntaxRule objects, OpenJava provides a class library of subclasses of SyntaxRule, such as parsers of regular Java syntax elements and synthesizing parser for tying, repeating or selecting other SyntaxRule

objects. Metaprogrammers can define their desired clauses by using this library or by implementing a new subclass of SyntaxRule.

3.7 Metaclass Model of OpenJava

A class must be managed by a single metaclass in OpenJava. Though it would be useful if programmers could apply several metaclasses to a class, we did not implement such a feature because there is a problem of conflict of translation between metaclasses. And, a metaclass for a class A does not manage a subclass A' of A, that is, the metaclass of A does not perform the callee-side and caller-side translation of A' it is not specified to be the metaclass of A' in the source program declaring A'.

For innerclasses such as member classes, local classes, anonymous classes in the Java language, each of them are also an instance of a metaclass in OpenJava. Thus programmers may apply a desired metaclass to such classes.

4 Related Work

There are a number of systems using the class metaobjects model for representing a logical structure of a program: 3-KRS[16], ObjVlisp[5], CLOS MOP[13], Smalltalk-80[8], and so on. The reflection API[11] of the Java language also uses this model although the reflection API does not allow to change class metaobjects; it only allows to inspect them. Furthermore, the reflection API uses class metaobjects for making class definition accessible at runtime. On the other hand, OpenJava uses class metaobjects for macro expansion at compile-time.

OpenC++[3] also uses the class metaobject model. OpenJava inherits several features, such as the type-driven translation mechanism, from OpenC++. However, the data structure mainly used in OpenC++ is still an AST (abstract syntax tree). MPC++[10] and EPP[9] are similar to OpenC++ with respect to the data structure. As mentioned in section 2, an AST is not an appropriate abstraction for some macros frequently used in OO programming.

5 Conclusion

This paper describes OpenJava, which is a macro system for Java providing a data structure called class metaobjects. A number of research activities have been done for enhancing expressive power of macro systems. This research is also in this stream. OpenJava is a macro system with a data structure representing a logical structure of an OO program. This made it easier to describe typical macros for OO programming which was difficult to describe with ordinary macro systems.

To show the effectiveness of OpenJava, we implemented some macros in OpenJava for supporting programming with design patterns. Although we saw that class metaobjects are useful for describing those macros, we also found

limitations of OpenJava. Since a single design pattern usually contains several classes, a macro system should be able to deal with those classes as a single entity[17]. However it is not easy for OpenJava to do that because macros are applied to each class. It is future work to address this problem by incorporate OpenJava with techniques like aspect-oriented programming[12].

References

[1] Bosch, J. Design patterns as language constructs. *Journal of Object Oriented Programming* (1997).

[2] Brown, P. J. *Macro Processors and Techniques for Portable Software.* Wiley, 1974.

[3] Chiba, S. A metaobject protocol for c++. *SIGPLAN Notices 30*, 10 (1995), 285–299.

[4] Chiba, S. Macro processing in object-oriented languages. In *Proceedings of TOOLS Pacific '98* (Australia, November 1998), IEEE, IEEE Press.

[5] Cointe, P. Metaclasses are first class : the objvlisp model. *SIGPLAN Notices 22*, 12 (December 1987), 156–162.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[7] Gil, J., and Lorenz, D. H. Design patterns and language design. *IEEE Computer 31*, 3 (March 1998), 118–120.

[8] Goldberg, A., and Robson, D. *Smalltalk-80: The Language.* Addison Wesley, 1989.

[9] Ichisugi, Y., and Roudier, Y. Extensible Java preprocessor kit and tiny data-parallel Java. In *Proceedings of ISCOPE'97* (California, December 1997).

[10] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., and Kubota, K. Design and implementation of metalevel architecture in c++ - mpc++ approach -. In *Proceedings of Reflection'96* (1996), pp. 153–166.

[11] JavaSoft. Java core reflection api and specification. online publishing, January 1997.

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., and Irwin, J. Aspect-oriented programming. *LNCS 1241* (June 1997), 220–242.

[13] Kiczales, G., Rivières, J., and Bobrow, D. G. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[14] Ladd, D. A., and Ramming, J. C. A* : A language for implementing language processors. *IEEE Transactions on Software Engineering 21*, 11 (November 1995), 894–901.

[15] Maddox, W. Semantically-sensitive macroprocessing. Master's thesis, University of California, Berkeley, 1989. ucb/csd 89/545.

[16] Maes, P. Concepts and experiments in computational reflection. *SIGPLAN Notices 22*, 12 (October 1987), 147–155.

[17] Soukup, J. Implementing patterns. In *Pattern Languages of Program Design.* Addison-Wesley, 1995, ch. 20, pp. 395–412.

[18] Steel Jr., G. L. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990.

[19] Weise, D., and Crew, R. Programmable syntax macros. *SIGPLAN Notices 28*, 6 (1993), 156–165.

Java™ Media Framework API Guide

November 19,1999
JMF 2.0 FCS

Contents

Preface

The Java™ Media Framework (JMF) is an application programming interface (API) for incorporating time-based media into Java applications and applets. This guide is intended for Java programmers who want to incorporate time-based media into their applications and for technology providers who are interested in extending JMF and providing JMF plug-ins to support additional media types and perform custom processing and rendering.

About JMF

The JMF 1.0 API (the Java Media Player API) enabled programmers to develop Java programs that presented time-based media. The JMF 2.0 API extends the framework to provide support for capturing and storing media data, controlling the type of processing that is performed during playback, and performing custom processing on media data streams. In addition, JMF 2.0 defines a plug-in API that enables advanced developers and technology providers to more easily customize and extend JMF functionality.

The following classes and interfaces are new in JMF 2.0:

`AudioFormat`	`BitRateControl`	`Buffer`
`BufferControl`	`BufferToImage`	`BufferTransferHandler`
`CaptureDevice`	`CaptureDeviceInfo`	`CaptureDeviceManager`
`CloneableDataSource`	`Codec`	`ConfigureCompleteEvent`
`ConnnectionErrorEvent`	`DataSink`	`DataSinkErrorEvent`
`DataSinkEvent`	`DataSinkListener`	`Demultiplexer`

`Effect`	`EndOfStreamEvent`	`FileTypeDescriptor`
`Format`	`FormatChangeEvent`	`FormatControl`
`FrameGrabbingControl`	`FramePositioningControl`	`FrameProcessingControl`
`FrameRateControl`	`H261Control`	`H261Format`
`H263Control`	`H263Format`	`ImageToBuffer`
`IndexedColorFormat`	`InputSourceStream`	`KeyFrameControl`
`MonitorControl`	`MpegAudioControl`	`Multiplexer`
`NoStorageSpaceErrorEvent`	`PacketSizeControl`	`PlugIn`
`PlugInManager`	`PortControl`	`Processor`
`ProcessorModel`	`PullBufferDataSource`	`PullBufferStream`
`PushBufferDataSource`	`PushBufferStream`	`QualityControl`
`Renderer`	`RGBFormat`	`SilenceSuppressionControl`
`StreamWriterControl`	`Track`	`TrackControl`
`VideoFormat`	`VideoRenderer`	`YUVFormat`

In addition, the `MediaPlayer` Java Bean has been included with the JMF API in `javax.media.bean.playerbean`. `MediaPlayer` can be instantiated directly and used to present one or more media streams.

Future versions of the JMF API will provide additional functionality and enhancements while maintaining compatibility with the current API.

Design Goals for the JMF API

JMF 2.0 supports media capture and addresses the needs of application developers who want additional control over media processing and rendering. It also provides a plug-in architecture that provides direct access to media data and enables JMF to be more easily customized and extended. JMF 2.0 is designed to:

- Be easy to program
- Support capturing media data
- Enable the development of media streaming and conferencing applications in Java

- Enable advanced developers and technology providers to implement custom solutions based on the existing API and easily integrate new features with the existing framework
- Provide access to raw media data
- Enable the development of custom, downloadable demultiplexers, codecs, effects processors, multiplexers, and renderers (JMF *plug-ins*)
- Maintain compatibility with JMF 1.0

About the JMF RTP APIs

The classes in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` provide support for RTP (Real-Time Transport Protocol). RTP enables the transmission and reception of real-time media streams across the network. RTP can be used for media-on-demand applications as well as interactive services such as Internet telephony.

JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

The first version of the JMF RTP APIs (referred to as the RTP Session Manager API) enabled developers to receive RTP streams and play them using JMF. In JMF 2.0, the RTP APIs also support the transmission of RTP streams.

The following RTP classes and interfaces are new in JMF 2.0:

`SendStream`	`SendStreamListener`	`InactiveSendStreamEvent`
`ActiveSendStreamEvent`	`SendPayloadChangeEvent`	`NewSendStreamEvent`
`GlobalTransmissionStats`	`TransmissionStats`	

The RTP packages have been reorganized and some classes, interfaces, and methods have been renamed to make the API easier to use. The package reorganization consists of the following changes:

- The RTP event classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.event`.
- The RTCP-related classes that were in `javax.media.rtp.session` are now in `javax.media.rtp.rtcp`.

- The rest of the classes in `javax.media.rtp.session` are now in `javax.media.rtp` and the `javax.media.rtp.session` package has been removed.

The name changes consist primarily of the removal of the RTP and RTCP prefixes from class and interface names and the elimination of non-standard abbreviations. For example, `RTPRecvStreamListener` has been renamed to `ReceiveStreamListener`. For a complete list of the changes made to the RTP packages, see the JMF 2.0 Beta release notes.

In addition, changes were made to the RTP APIs to make them compatible with other changes in JMF 2.0:

- `javax.media.rtp.session.io` and `javax.media.rtp.session.depacketizer` have been removed. Custom RTP packetizers and depacketizers are now supported through the JMF 2.0 plug-in architecture. Existing depacketizers will need to be ported to the new plug-in architecture.
- `Buffer` is now the basic unit of transfer between the `SessionManager` and other JMF objects, in place of `DePacketizedUnit` and `DePacketizedObject`. RTP-formatted `Buffers` have a specific format for their data and header objects.
- `BaseEncodingInfo` has been replaced by the generic JMF `Format` object. An RTP-specific `Format` is differentiated from other formats by its encoding string. Encoding strings for RTP-specific `Formats` end in `_RTP`. Dynamic payload information can be provided by associating a dynamic payload number with a `Format` object.

Design Goals for the JMF RTP APIs

The RTP APIs in JMF 2.0 support the reception and transmission of RTP streams and address the needs of application developers who want to use RTP to implement media streaming and conferencing applications. These APIs are designed to:

- Enable the development of media streaming and conferencing applications in Java
- Support media data reception and transmission using RTP and RTCP
- Support custom packetizer and depacketizer plug-ins through the JMF 2.0 plug-in architecture.
- Be easy to program

Partners in the Development of the JMF API

The JMF 2.0 API is being jointly designed by Sun Microsystems, Inc. and IBM Corporation.

The JMF 1.0 API was jointly developed by Sun Microsystems Inc., Intel Corporation, and Silicon Graphics, Inc.

Contact Information

For the latest information about JMF, visit the Sun Microsystems, Inc. website at:

```
http://java.sun.com/products/java-media/jmf/
```

Additional information about JMF can be found on the IBM Corporation website at:

```
http://www.software.ibm.com/net.media/
```

About this Document

This document describes the architecture and use of the JMF 2.0 API. It replaces the Java Media Player Guide distributed in conjunction with the JMF 1.0 releases.

Except where noted, the information in this book is not implementation specific. For examples specific to the JMF reference implementation developed by Sun Microsystems and IBM corporation, see the sample code and solutions available from Sun's JMF website (`http://java.sun.com/products/java-media/jmf/index.html`).

Guide to Contents

This document is split into two parts:

- Part 1 describes the features provided by the JMF 2.0 API and illustrates how you can use JMF to incorporate time-based media in your Java applications and applets.
- Part 2 describes the support for real-time streaming provided by the JMF RTP APIs and illustrates how to send and receive streaming media across the network.

Part 1 is organized into six chapters:

- **"Working with Time-Based Media"**—sets the stage for JMF by introducing the key concepts of media content, presentation, processing, and recording.
- **"Understanding JMF"**—introduces the JMF 2.0 API and describes the high-level architecture of the framework.
- **"Presenting Time-Based Media with JMF"**—describes how to use JMF `Players` and `Processors` to present time-based media.
- **"Processing Time-Based Media with JMF"**—describes how to manipulate media data using a JMF `Processor`.
- **"Capturing Time-Based Media with JMF"**—describes how to record media data using JMF `DataSources` and `Processors`.
- **"Extending JMF"**—describes how to enhance JMF functionality by creating new processing plug-ins and implementing custom JMF classes.

Part 2 is organized into six chapters:

- **"Working with Real-Time Media Streams"**–provides an overview of streaming media and the Real-time Transport protocol (RTP).
- **"Understanding the JMF RTP API"**—describes the JMF RTP APIs.
- **"Receiving and Presenting RTP Media Streams"**—illustrates how to handle RTP Client operations.
- **"Transmitting RTP Media Streams"**—illustrates how to handle RTP Server operations.
- **"Importing and Exporting RTP Media Streams"**—shows how to read and write RTP data to a file.
- **"Creating Custom Packetizers and Depacketizers"**—describes how to use JMF plug-ins to support additional RTP packet formats and codecs.

At the end of this document, you'll find Appendices that contain complete sample code for some of the examples used in these chapters and a glossary of JMF-specific terms.

Change History

Version JMF 2.0 FCS

- Fixed references to `TrackControl` methods to reflect modified `TrackControl` API.
- Fixed minor sample code errors.
- Clarified behavior of cloneable data sources.
- Clarified order of events when writing to a file.

Version 0.9

Internal Review Draft

Version 0.8

JMF 2.0 Beta draft:

- Added an introduction to RTP, Working with Real-Time Media Streams, and updated the RTP chapters.
- Updated to reflect API changes since the Early Access release.
- Added an example of registering a plug-in with the `PlugInManager`.
- Added chapter, figure, table, and example numbers and changed the example code style.

Version 0.7

JMF 2.0 Early Access Release 1 draft:

- Updated and expanded RTP chapters in Part 2.
- Added Demultiplexer example to "Extending JMF".
- Updated to reflect API changes since the public review.

Version 0.6

Internal Review Draft

Version 0.5

JMF 2.0 API public review draft.

- Added new concepts chapter, **"Working with Time-Based Media"**.
- Reorganized architecture information in **"Understanding JMF"**.

- Incorporated RTP Guide as Part 2.

Version 0.4

JMF 2.0 API licensee review draft.

Comments

Please submit any comments or suggestions you have for improving this document to `jmf-comments@eng.sun.com`.

Part 1:
Java™ Media Framework

1

Working with Time-Based Media

Any data that changes meaningfully with respect to time can be characterized as time-based media. Audio clips, MIDI sequences, movie clips, and animations are common forms of time-based media. Such media data can be obtained from a variety of sources, such as local or network files, cameras, microphones, and live broadcasts.

This chapter describes the key characteristics of time-based media and describes the use of time-based media in terms of a fundamental data processing model:

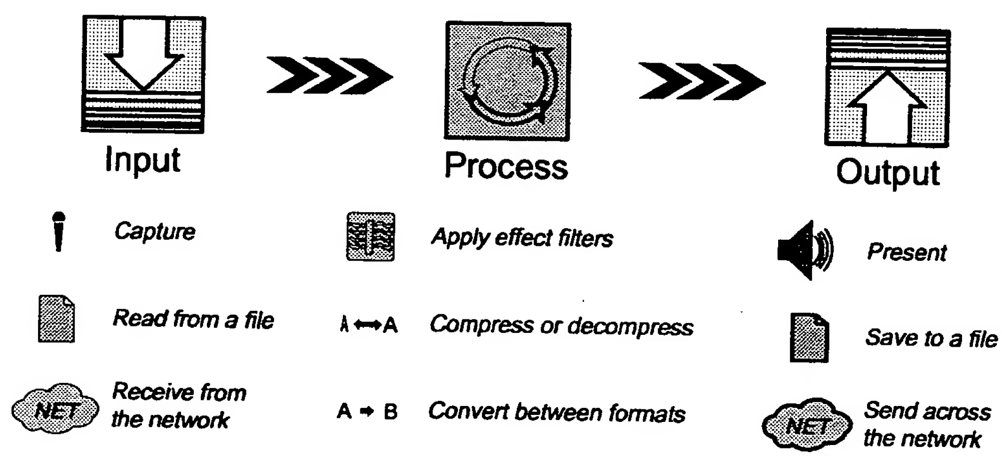


Figure 1-1: Media processing model.

Streaming Media

A key characteristic of time-based media is that it requires timely delivery and processing. Once the flow of media data begins, there are strict timing deadlines that must be met, both in terms of receiving and presenting the data. For this reason, time-based media is often referred to as *streaming media*—it is delivered in a steady stream that must be received and processed within a particular timeframe to produce acceptable results.

For example, when a movie is played, if the media data cannot be delivered quickly enough, there might be odd pauses and delays in playback. On the other hand, if the data cannot be received and processed quickly enough, the movie might appear jumpy as data is lost or frames are intentionally dropped in an attempt to maintain the proper playback rate.

Content Type

The format in which the media data is stored is referred to as its *content type*. QuickTime, MPEG, and WAV are all examples of content types. Content type is essentially synonymous with file type—content type is used because media data is often acquired from sources other than local files.

Media Streams

A *media stream* is the media data obtained from a local file, acquired over the network, or captured from a camera or microphone. Media streams often contain multiple channels of data called *tracks*. For example, a Quicktime file might contain both an audio track and a video track. Media streams that contain multiple tracks are often referred to as *multiplexed* or *complex* media streams. *Demultiplexing* is the process of extracting individual tracks from a complex media stream.

A track's *type* identifies the kind of data it contains, such as audio or video. The *format* of a track defines how the data for the track is structured.

A media stream can be identified by its location and the protocol used to access it. For example, a URL might be used to describe the location of a QuickTime file on a local or remote system. If the file is local, it can be accessed through the FILE protocol. On the other hand, if it's on a web server, the file can be accessed through the HTTP protocol. A *media locator* provides a way to identify the location of a media stream when a URL can't be used.

Media streams can be categorized according to how the data is delivered:

- Pull—data transfer is initiated and controlled from the client side. For example, Hypertext Transfer Protocol (HTTP) and FILE are pull protocols.
- Push—the server initiates data transfer and controls the flow of data. For example, Real-time Transport Protocol (RTP) is a push protocol used for streaming media. Similarly, the SGI MediaBase protocol is a push protocol used for video-on-demand (VOD).

Common Media Formats

The following tables identify some of the characteristics of common media formats. When selecting a format, it's important to take into account the characteristics of the format, the target environment, and the expectations of the intended audience. For example, if you're delivering media content via the web, you need to pay special attention to the bandwidth requirements.

The CPU Requirements column characterizes the processing power necessary for optimal presentation of the specified format. The Bandwidth Requirements column characterizes the transmission speeds necessary to send or receive data quickly enough for optimal presentation.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Cinepak	AVI QuickTime	Medium	Low	High
MPEG-1	MPEG	High	High	High
H.261	AVI RTP	Low	Medium	Medium
H.263	QuickTime AVI RTP	Medium	Medium	Low
JPEG	QuickTime AVI RTP	High	High	High

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
Indeo	QuickTime AVI	Medium	Medium	Medium

Table 1-1: Common video formats.

Format	Content Type	Quality	CPU Requirements	Bandwidth Requirements
PCM	AVI QuickTime WAV	High	Low	High
Mu-Law	AVI QuickTime WAV RTP	Low	Low	High
ADPCM (DVI, IMA4)	AVI QuickTime WAV RTP	Medium	Medium	Medium
MPEG-1	MPEG	High	High	High
MPEG Layer3	MPEG	High	High	Medium
GSM	WAV RTP	Low	Low	Low
G.723.1	WAV RTP	Medium	Medium	Low

Table 1-2: Common audio formats.

Some formats are designed with particular applications and requirements in mind. High-quality, high-bandwidth formats are generally targeted toward CD-ROM or local storage applications. H.261 and H.263 are generally used for video conferencing applications and are optimized for video where there's not a lot of action. Similarly, G.723 is typically used to produce low bit-rate speech for telephony applications.

Media Presentation

Most time-based media is audio or video data that can be presented through output devices such as speakers and monitors. Such devices are the most common *destination* for media data output. Media streams can also be sent to other destinations—for example, saved to a file or transmitted across the network. An output destination for media data is sometimes referred to as a *data sink*.

Presentation Controls

While a media stream is being presented, VCR-style presentation controls are often provided to enable the user to control playback. For example, a control panel for a movie player might offer buttons for stopping, starting, fast-forwarding, and rewinding the movie.

Latency

In many cases, particularly when presenting a media stream that resides on the network, the presentation of the media stream cannot begin immediately. The time it takes before presentation can begin is referred to as the *start latency*. Users might experience this as a delay between the time that they click the start button and the time when playback actually starts.

Multimedia presentations often combine several types of time-based media into a synchronized presentation. For example, background music might be played during an image slide-show, or animated text might be synchronized with an audio or video clip. When the presentation of multiple media streams is synchronized, it is essential to take into account the start latency of each stream—otherwise the playback of the different streams might actually begin at different times.

Presentation Quality

The quality of the presentation of a media stream depends on several factors, including:

- The compression scheme used
- The processing capability of the playback system
- The bandwidth available (for media streams acquired over the network)

Traditionally, the higher the quality, the larger the file size and the greater the processing power and bandwidth required. Bandwidth is usually represented as the number of bits that are transmitted in a certain period of time—the *bit rate.*

To achieve high-quality video presentations, the number of frames displayed in each period of time (the *frame rate*) should be as high as possible. Usually movies at a frame rate of 30 frames-per-second are considered indistinguishable from regular TV broadcasts or video tapes.

Media Processing

In most instances, the data in a media stream is manipulated before it is presented to the user. Generally, a series of processing operations occur before presentation:

1. If the stream is multiplexed, the individual tracks are extracted.
2. If the individual tracks are compressed, they are decoded.
3. If necessary, the tracks are converted to a different format.
4. Effect filters are applied to the decoded tracks (if desired).

The tracks are then delivered to the appropriate output device. If the media stream is to be stored instead of rendered to an output device, the processing stages might differ slightly. For example, if you wanted to capture audio and video from a video camera, process the data, and save it to a file:

1. The audio and video tracks would be captured.
2. Effect filters would be applied to the raw tracks (if desired).
3. The individual tracks would be encoded.
4. The compressed tracks would be multiplexed into a single media stream.
5. The multiplexed media stream would then be saved to a file.

Demultiplexers and Multiplexers

A demultiplexer extracts individual tracks of media data from a multiplexed media stream. A *mutliplexer* performs the opposite function, it takes individual tracks of media data and merges them into a single multiplexed media stream.

Codecs

A codec performs media-data compression and decompression. When a track is encoded, it is converted to a compressed format suitable for storage or transmission; when it is decoded it is converted to a non-compressed (raw) format suitable for presentation.

Each codec has certain input formats that it can handle and certain output formats that it can generate. In some situations, a series of codecs might be used to convert from one format to another.

Effect Filters

An effect filter modifies the track data in some way, often to create special effects such as blur or echo.

Effect filters are classified as either pre-processing effects or post-processing effects, depending on whether they are applied before or after the codec processes the track. Typically, effect filters are applied to uncompressed (raw) data.

Renderers

A renderer is an abstraction of a presentation device. For audio, the presentation device is typically the computer's hardware audio card that outputs sound to the speakers. For video, the presentation device is typically the computer monitor.

Compositing

Certain specialized devices support *compositing*. Compositing time-based media is the process of combining multiple tracks of data onto a single presentation medium. For example, overlaying text on a video presentation is one common form of compositing. Compositing can be done in either hardware or software. A device that performs compositing can be abstracted as a renderer that can receive multiple tracks of input data.

Media Capture

Time-based media can be captured from a live source for processing and playback. For example, audio can be captured from a microphone or a video capture card can be used to obtain video from a camera. Capturing can be thought of as the *input* phase of the standard media processing model.

A capture device might deliver multiple media streams. For example, a video camera might deliver both audio and video. These streams might be captured and manipulated separately or combined into a single, multiplexed stream that contains both an audio track and a video track.

Capture Devices

To capture time-based media you need specialized hardware—for example, to capture audio from a live source, you need a microphone and an appropriate audio card. Similarly, capturing a TV broadcast requires a TV tuner and an appropriate video capture card. Most systems provide a query mechanism to find out what capture devices are available.

Capture devices can be characterized as either push or pull sources. For example, a still camera is a pull source—the user controls when to capture an image. A microphone is a push source—the live source continuously provides a stream of audio.

The format of a captured media stream depends on the processing performed by the capture device. Some devices do very little processing and deliver raw, uncompressed data. Other capture devices might deliver the data in a compressed format.

Capture Controls

Controls are sometimes provided to enable the user to manage the capture process. For example, a capture control panel might enable the user to specify the data rate and encoding type for the captured stream and start and stop the capture process.

2

Understanding JMF

Java™ Media Framework (JMF) provides a unified architecture and messaging protocol for managing the acquisition, processing, and delivery of time-based media data. JMF is designed to support most standard media content types, such as AIFF, AU, AVI, GSM, MIDI, MPEG, QuickTime, RMF, and WAV.

By exploiting the advantages of the Java platform, JMF delivers the promise of "Write Once, Run Anywhere™" to developers who want to use media such as audio and video in their Java programs. JMF provides a common cross-platform Java API for accessing underlying media frameworks. JMF implementations can leverage the capabilities of the underlying operating system, while developers can easily create portable Java programs that feature time-based media by writing to the JMF API.

With JMF, you can easily create applets and applications that present, capture, manipulate, and store time-based media. The framework enables advanced developers and technology providers to perform custom processing of raw media data and seamlessly extend JMF to support additional content types and formats, optimize handling of supported formats, and create new presentation mechanisms.

High-Level Architecture

Devices such as tape decks and VCRs provide a familiar model for recording, processing, and presenting time-based media. When you play a movie using a VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads and interprets the data on the tape and sends appropriate signals to your television and speakers.

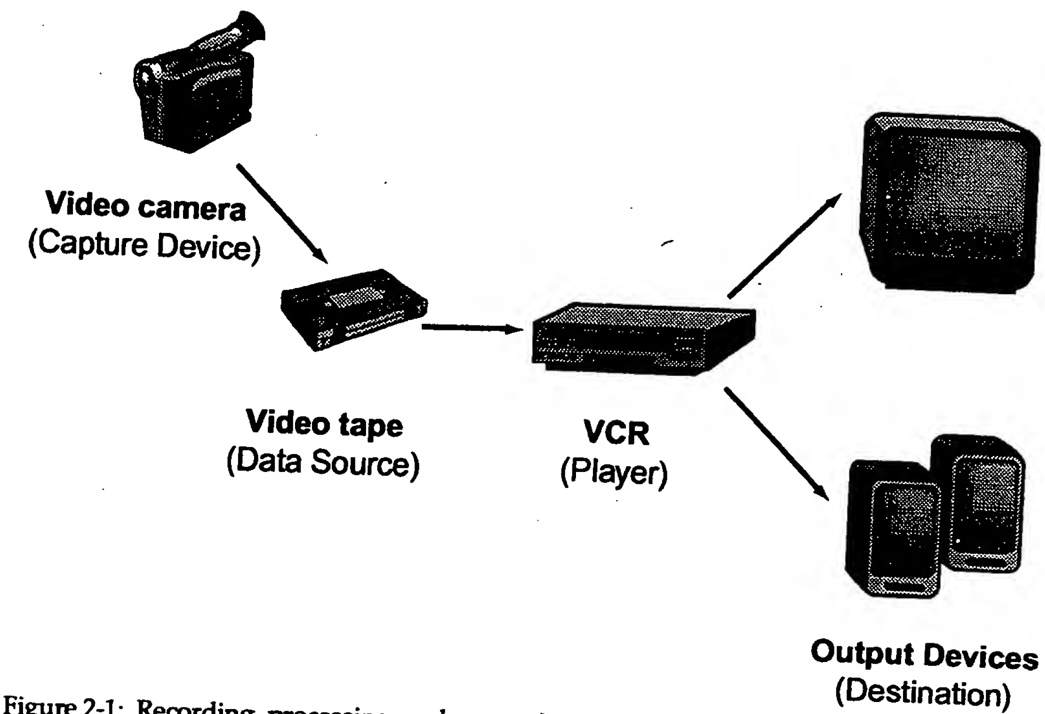


Figure 2-1: Recording, processing, and presenting time-based media.

JMF uses this same basic model. A *data source* encapsulates the media stream much like a video tape and a *player* provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.

Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media. JMF also provides a lower-level API that supports the seamless integration of custom processing components and extensions. This layering provides Java developers with an easy-to-use API for incorporating time-based media into Java programs while maintaining the flexibility and extensibility required to support advanced media applications and future media technologies.

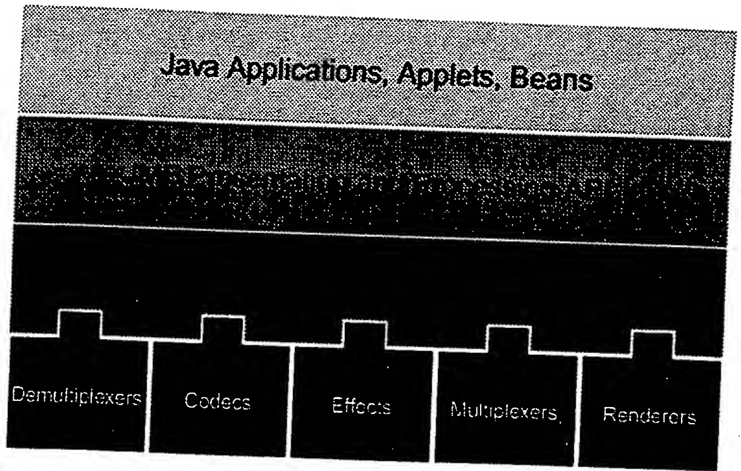


Figure 2-2: High-level JMF achitecture.

Time Model

JMF keeps time to nanosecond precision. A particular point in time is typically represented by a `Time` object, though some classes also support the specification of time in nanoseconds.

Classes that support the JMF time model implement `Clock` to keep track of time for a particular media stream. The `Clock` interface defines the basic timing and synchronization operations that are needed to control the presentation of media data.

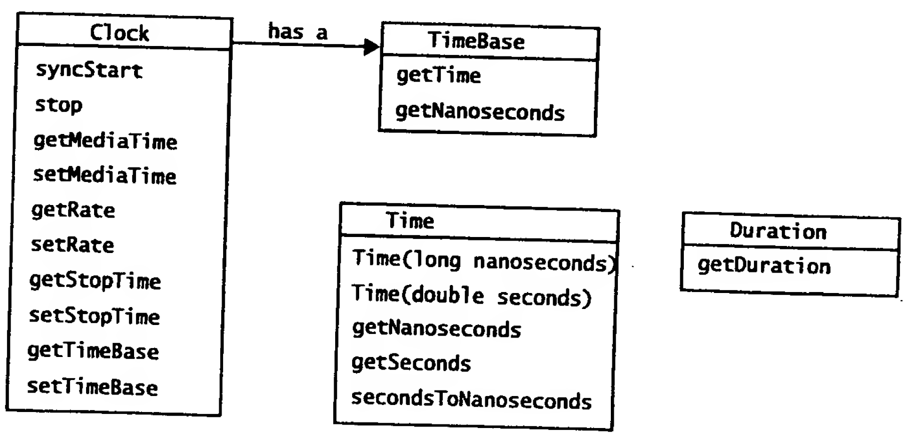


Figure 2-3: JMF time model.

A `Clock` uses a `TimeBase` to keep track of the passage of time while a media stream is being presented. A `TimeBase` provides a constantly ticking time source, much like a crystal oscillator in a watch. The only information that a `TimeBase` provides is its current time, which is referred to as the *time-base*

time. The time-base time cannot be stopped or reset. Time-base time is often based on the system clock.

A `Clock` object's *media time* represents the current position within a media stream—the beginning of the stream is media time zero, the end of the stream is the maximum media time for the stream. The *duration* of the media stream is the elapsed time from start to finish—the length of time that it takes to present the media stream. (Media objects implement the `Duration` interface if they can report a media stream's duration.)

To keep track of the current media time, a `Clock` uses:

- The time-base start-time—the time that its `TimeBase` reports when the presentation begins.
- The media start-time—the position in the media stream where presentation begins.
- The playback rate—how fast the `Clock` is running in relation to its `TimeBase`. The *rate* is a scale factor that is applied to the `TimeBase`. For example, a rate of 1.0 represents the normal playback rate for the media stream, while a rate of 2.0 indicates that the presentation will run at twice the normal rate. A negative rate indicates that the `Clock` is running in the opposite direction from its `TimeBase`—for example, a negative rate might be used to play a media stream backward.

When presentation begins, the media time is mapped to the time-base time and the advancement of the time-base time is used to measure the passage of time. During presentation, the current media time is calculated using the following formula:

```
MediaTime = MediaStartTime + Rate(TimeBaseTime - TimeBaseStartTime)
```

When the presentation stops, the media time stops, but the time-base time continues to advance. If the presentation is restarted, the media time is remapped to the current time-base time.

Managers

The JMF API consists mainly of interfaces that define the behavior and interaction of objects used to capture, process, and present time-based media. Implementations of these interfaces operate within the structure of the framework. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes.

JMF uses four managers:

- Manager—handles the construction of Players, Processors, DataSources, and DataSinks. This level of indirection allows new implementations to be integrated seamlessly with JMF. From the client perspective, these objects are always created the same way whether the requested object is constructed from a default implementation or a custom one.
- PackageManager—maintains a registry of packages that contain JMF classes, such as custom Players, Processors, DataSources, and DataSinks.
- CaptureDeviceManager—maintains a registry of available capture devices.
- PlugInManager—maintains a registry of available JMF plug-in processing components, such as Multiplexers, Demultiplexers, Codecs, Effects, and Renderers.

To write programs based on JMF, you'll need to use the Manager create methods to construct the Players, Processors, DataSources, and DataSinks for your application. If you're capturing media data from an input device, you'll use the CaptureDeviceManager to find out what devices are available and access information about them. If you're interested in controlling what processing is performed on the data, you might also query the PlugInManager to determine what plug-ins have been registered.

If you extend JMF functionality by implementing a new plug-in, you can register it with the PlugInManager to make it available to Processors that support the plug-in API. To use a custom Player, Processor, DataSource, or DataSink with JMF, you register your unique package prefix with the PackageManager.

Event Model

JMF uses a structured event reporting mechanism to keep JMF-based programs informed of the current state of the media system and enable JMF-based programs to respond to media-driven error conditions, such as out-of data and resource unavailable conditions. Whenever a JMF object needs to report on the current conditions, it posts a MediaEvent. MediaEvent is subclassed to identify many particular types of events. These objects follow the established Java Beans patterns for events.

For each type of JMF object that can post MediaEvents, JMF defines a corresponding listener interface. To receive notification when a MediaEvent is posted, you implement the appropriate listener interface and register your listener class with the object that posts the event by calling its addListener method.

Controller objects (such as Players and Processors) and certain Control objects such as GainControl post media events.

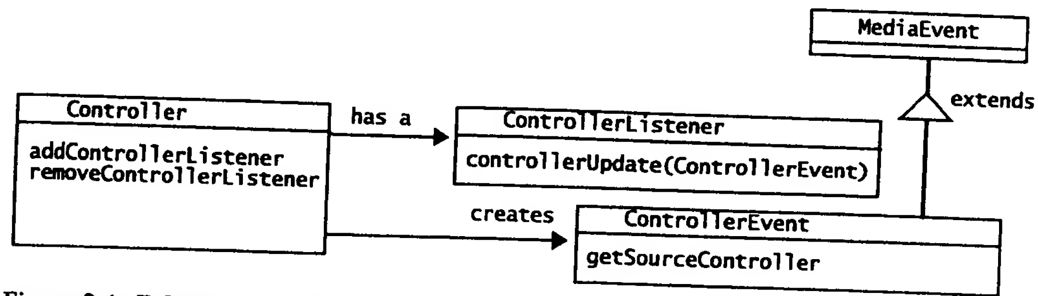


Figure 2-4: JMF event model.

RTPSessionManager objects also post events. For more information, see "RTP Events" on page 122.

Data Model

JMF media players usually use DataSources to manage the transfer of media-content. A DataSource encapsulates both the location of media and the protocol and software used to deliver the media. Once obtained, the source cannot be reused to deliver other media.

A DataSource is identified by either a JMF MediaLocator or a URL (universal resource locator). A MediaLocator is similar to a URL and can be constructed from a URL, but can be constructed even if the corresponding protocol handler is not installed on the system. (In Java, a URL can only be constructed if the corresponding protocol handler is installed on the system.)

A DataSource manages a set of SourceStream objects. A standard data source uses a byte array as the unit of transfer. A *buffer data source* uses a Buffer object as its unit of transfer. JMF defines several types of DataSource objects:

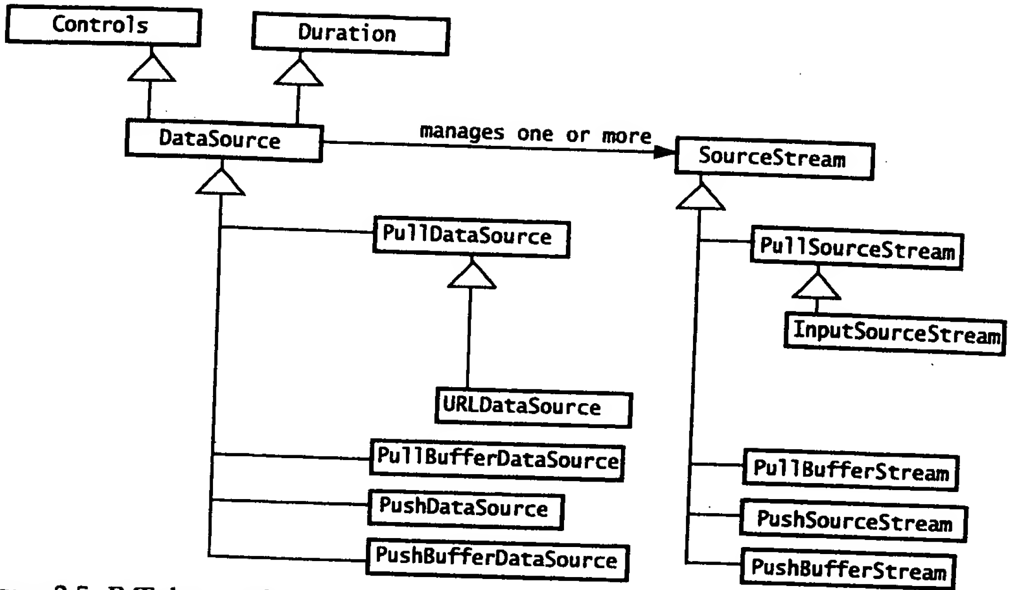


Figure 2-5: JMF data model.

Push and Pull Data Sources

Media data can be obtained from a variety of sources, such as local or network files and live broadcasts. JMF data sources can be categorized according to how data transfer is initiated:

- *Pull Data-Source*—the client initiates the data transfer and controls the flow of data from pull data-sources. Established protocols for this type of data include Hypertext Transfer Protocol (HTTP) and FILE. JMF defines two types of pull data sources: `PullDataSource` and `PullBufferDataSource`, which uses a `Buffer` object as its unit of transfer.
- *Push Data-Source*—the server initiates the data transfer and controls the flow of data from a push data-source. Push data-sources include broadcast media, multicast media, and video-on-demand (VOD). For broadcast data, one protocol is the Real-time Transport Protocol (RTP), under development by the Internet Engineering Task Force (IETF). The MediaBase protocol developed by SGI is one protocol used for VOD. JMF defines two types of push data sources: `PushDataSource` and `PushBufferDataSource`, which uses a `Buffer` object as its unit of transfer.

The degree of control that a client program can extend to the user depends on the type of data source being presented. For example, an MPEG file can

be repositioned and a client program could allow the user to replay the video clip or seek to a new position in the video. In contrast, broadcast media is under server control and cannot be repositioned. Some VOD protocols might support limited user control—for example, a client program might be able to allow the user to seek to a new position, but not fast forward or rewind.

Specialty DataSources

JMF defines two types of specialty data sources, cloneable data sources and merging data sources.

A cloneable data source can be used to create clones of either a pull or push `DataSource`. To create a cloneable `DataSource`, you call the `Manager` `createCloneableDataSource` method and pass in the `DataSource` you want to clone. Once a `DataSource` has been passed to `createCloneableDataSource`, you should only interact with the cloneable `DataSource` and its clones; the original `DataSource` should no longer be used directly.

Cloneable data sources implement the `SourceCloneable` interface, which defines one method, `createClone`. By calling `createClone`, you can create any number of clones of the `DataSource that was` used to construct the cloneable `DataSource`. The clones can be controlled through the cloneable `DataSource` used to create them— when `connect`, `disconnect`, `start`, or `stop` is called on the cloneable `DataSource`, the method calls are propagated to the clones.

The clones don't necessarily have the same properties as the cloneable data source used to create them or the original `DataSource`. For example, a cloneable data source created for a capture device might function as a master data source for its clones—in this case, unless the cloneable data source is used, the clones won't produce any data. If you hook up both the cloneable data source and one or more clones, the clones will produce data at the same rate as the master.

A `MergingDataSource` can be used to combine the `SourceStreams` from several `DataSources` into a single `DataSource`. This enables a set of `DataSources` to be managed from a single point of control—when `connect`, `disconnect`, `start`, or `stop` is called on the `MergingDataSource`, the method calls are propagated to the merged `DataSources`.

To construct a `MergingDataSource`, you call the `Manager` `createMergingDataSource` method and pass in an array that contains the data sources you want to merge. To be merged, all of the `DataSources` must be of the

same type; for example, you cannot merge a `PullDataSource` and a `PushDataSource`. The duration of the merged `DataSource` is the maximum of the merged `DataSource` objects' durations. The `ContentType` is `application/mixed-media`.

Data Formats

The exact media format of an object is represented by a `Format` object. The format itself carries no encoding-specific parameters or global timing information, it describes the format's encoding name and the type of data the format requires.

JMF extends `Format` to define audio- and video-specific formats.

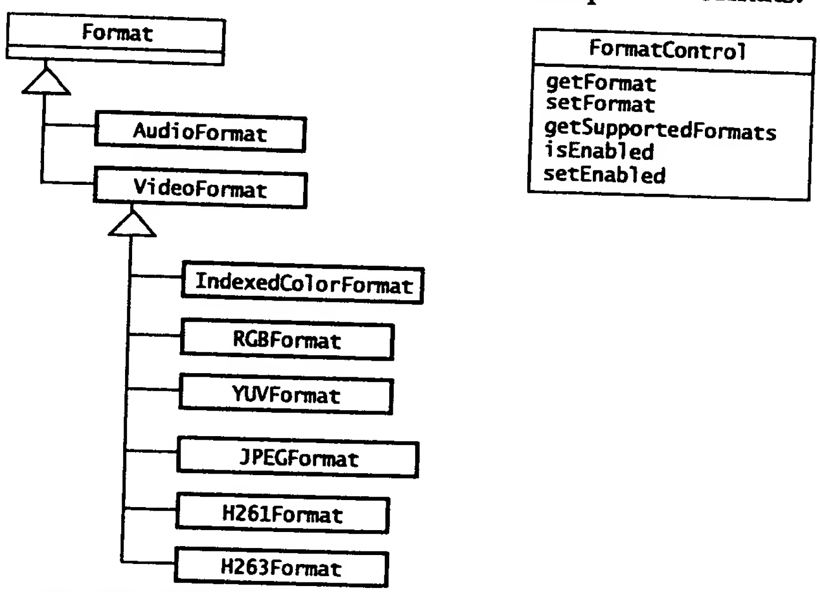


Figure 2-6: JMF media formats.

An `AudioFormat` describes the attributes specific to an audio format, such as sample rate, bits per sample, and number of channels. A `VideoFormat` encapsulates information relevant to video data. Several formats are derived from `VideoFormat` to describe the attributes of common video formats, including:

- `IndexedColorFormat`
- `RGBFormat`
- `YUVFormat`
- `JPEGFormat`
- `H261Format`
- `H263Format`

To receive notification of format changes from a `Controller`, you implement the `ControllerListener` interface and listen for `FormatChangeEvents`. (For more information, see "Responding to Media Events" on page 54.)

Controls

JMF `Control` provides a mechanism for setting and querying attributes of an object. A `Control` often provides access to a corresponding user interface component that enables user control over an object's attributes. Many JMF objects expose `Controls`, including `Controller` objects, `DataSource` objects, `DataSink` objects, and JMF plug-ins.

Any JMF object that wants to provide access to its corresponding `Control` objects can implement the `Controls` interface. `Controls` defines methods for retrieving associated `Control` objects. `DataSource` and `PlugIn` use the `Controls` interface to provide access to their `Control` objects.

Standard Controls

JMF defines the standard `Control` interfaces shown in Figure 2-8:, "JMF controls."

`CachingControl` enables download progress to be monitored and displayed. If a `Player` or `Processor` can report its download progress, it implements this interface so that a progress bar can be displayed to the user.

`GainControl` enables audio volume adjustments such as setting the level and muting the output of a `Player` or `Processor`. It also supports a listener mechanism for volume changes.

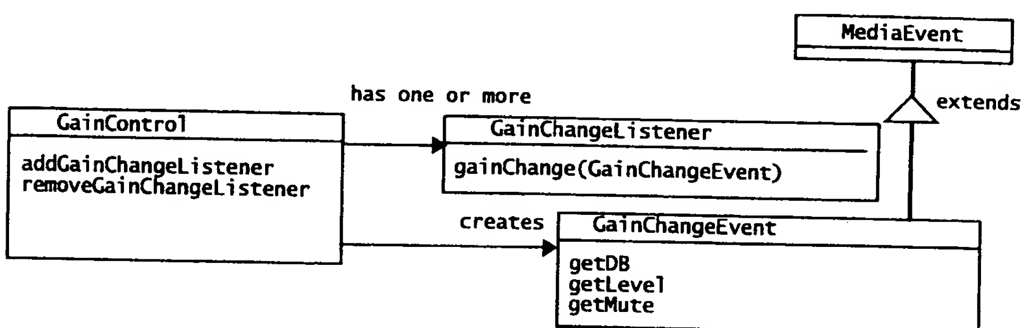


Figure 2-7: Gain control.

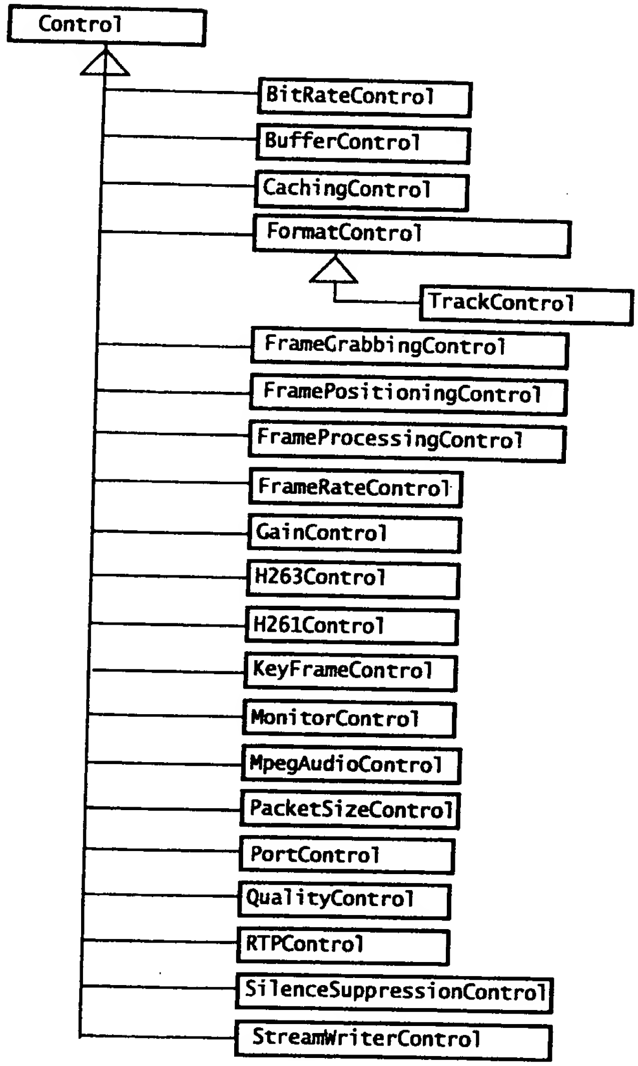


Figure 2-8: JMF controls.

`DataSink` or `Multiplexer` objects that read media from a `DataSource` and write it out to a destination such as a file can implement the `StreamWriterControl` interface. This `Control` enables the user to limit the size of the stream that is created.

`FramePositioningControl` and `FrameGrabbingControl` export frame-based capabilities for `Players` and `Processors`. `FramePositioningControl` enables precise frame positioning within a `Player` or `Processor` object's media stream. `FrameGrabbingControl` provides a mechanism for grabbing a still video frame from the video stream. The `FrameGrabbingControl` can also be supported at the `Renderer` level.

Objects that have a Format can implement the FormatControl interface to provide access to the Format. FormatControl also provides methods for querying and setting the format.

A TrackControl is a type of FormatControl that provides the mechanism for controlling what processing a Processor object performs on a particular track of media data. With the TrackControl methods, you can specify what format conversions are performed on individual tracks and select the Effect, Codec, or Renderer plug-ins that are used by the Processor. (For more information about processing media data, see "Processing Time-Based Media with JMF" on page 71.)

Two controls, PortControl and MonitorControl enable user control over the capture process. PortControl defines methods for controlling the output of a capture device. MonitorControl enables media data to be previewed as it is captured or encoded.

BufferControl enables user-level control over the buffering done by a particular object.

JMF also defines several codec controls to enable control over hardware or software encoders and decoders:

- BitRateControl—used to export the bit rate information for an incoming stream or to control the encoding bit rate. Enables specification of the bit rate in bits per second.
- FrameProcessingControl—enables the specification of frame processing parameters that allow the codec to perform minimal processing when it is falling behind on processing the incoming data.
- FrameRateControl—enables modification of the frame rate.
- H261Control—enables control over the H.261 video codec still-image transmission mode.
- H263Control—enables control over the H.263 video-codec parameters, including support for the unrestricted vector, arithmetic coding, advanced prediction, PB Frames, and error compensation extensions.
- KeyFrameControl—enables the specification of the desired interval between key frames. (The encoder can override the specified key-frame interval if necessary.)
- MpegAudioControl—exports an MPEG audio codec's capabilities and enables the specification of selected MPEG encoding parameters.
- QualityControl—enables specification of a preference in the trade-off

between quality and CPU usage in the processing performed by a codec. This quality hint can have different effects depending on the type of compression. A higher quality setting will result in better quality of the resulting bits, for example better image quality for video.

- `SilenceSuppressionControl`—enables specification of silence suppression parameters for audio codecs. When silence suppression mode is on, an audio encoder does not output any data if it detects silence at its input.

User Interface Components

A `Control` can provide access to a user interface `Component` that exposes its control behavior to the end user. To get the default user interface component for a particular `Control`, you call `getControlComponent`. This method returns an AWT `Component` that you can add to your applet's presentation space or application window.

A `Controller` might also provide access to user interface `Components`. For example, a `Player` provides access to both a visual component and a control panel component—to retrieve these components, you call the `Player` methods `getVisualComponent` and `getControlPanelComponent`.

If you don't want to use the default control components provided by a particular implementation, you can implement your own and use the event listener mechanism to determine when they need to be updated. For example, you might implement your own GUI components that support user interaction with a `Player`. Actions on your GUI components would trigger calls to the appropriate `Player` methods, such as `start` and `stop`. By registering your custom GUI components as `ControllerListeners` for the `Player`, you can also update your GUI in response to changes in the `Player` object's state.

Extensibility

Advanced developers and technology providers can extend JMF functionality in two ways:

- By implementing custom processing components (*plug-ins*) that can be interchanged with the standard processing components used by a JMF `Processor`

- By directly implementing the `Controller`, `Player`, `Processor`, `DataSource`, or `DataSink` interfaces

Implementing a JMF plug-in enables you to customize or extend the capabilities of a `Processor` without having to implement one from scratch. Once a plug-in is registered with JMF, it can be selected as a processing option for any `Processor` that supports the plug-in API. JMF plug-ins can be used to:

- Extend or replace a `Processor` object's processing capability piecewise by selecting the individual plug-ins to be used.
- Access the media data at specific points in the data flow. For example, different `Effect` plug-ins can be used for pre- and post-processing of the media data associated with a `Processor`.
- Process media data outside of a `Player` or `Processor`. For example, you might use a `Demultiplexer` plug-in to get individual audio tracks from a multiplexed media-stream so you could play the tracks through Java Sound.

In situations where an even greater degree of flexibility and control is required, custom implementations of the JMF `Controller`, `Player`, `Processor`, `DataSource`, or `DataSink` interfaces can be developed and used seamlessly with existing implementations. For example, if you have a hardware MPEG decoder, you might want to implement a `Player` that takes input from a `DataSource` and uses the decoder to perform the parsing, decoding, and rendering all in one step. Custom `Players` and `Processors` can also be implemented to integrate media engines such as Microsoft's Media Player, Real Network's RealPlayer, and IBM's HotMedia with JMF.

Note: JMF `Players` and `Processors` are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based `Players` and some `Processor` implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Presentation

In JMF, the presentation process is modeled by the `Controller` interface. `Controller` defines the basic state and control mechanism for an object that controls, presents, or captures time-based media. It defines the phases

that a media controller goes through and provides a mechanism for controlling the transitions between those phases. A number of the operations that must be performed before media data can be presented can be time consuming, so JMF allows programmatic control over when they occur.

A `Controller` posts a variety of controller-specific `MediaEvents` to provide notification of changes in its status. To receive events from a `Controller` such as a `Player`, you implement the `ControllerListener` interface. For more information about the events posted by a `Controller`, see "Controller Events" on page 30.

The JMF API defines two types of `Controllers`: `Players` and `Processors`. A `Player` or `Processor` is constructed for a particular data source and is normally not re-used to present other media data.

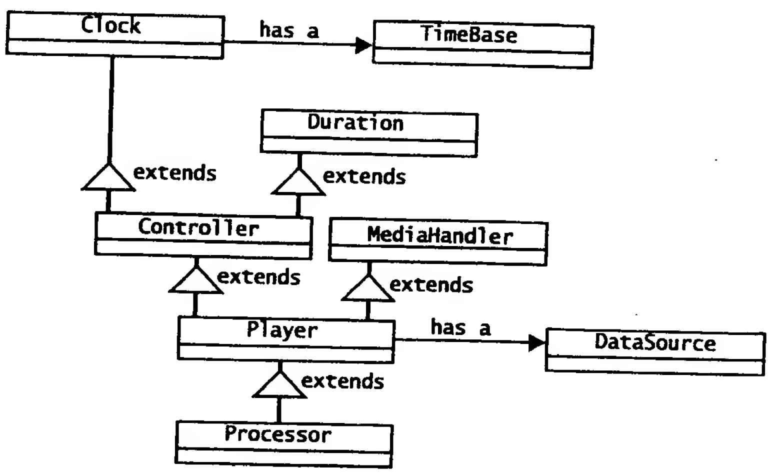


Figure 2-9: JMF controllers.

Players

A `Player` processes an input stream of media data and renders it at a precise time. A `DataSource` is used to deliver the input media-stream to the `Player`.The rendering destination depends on the type of media being presented.

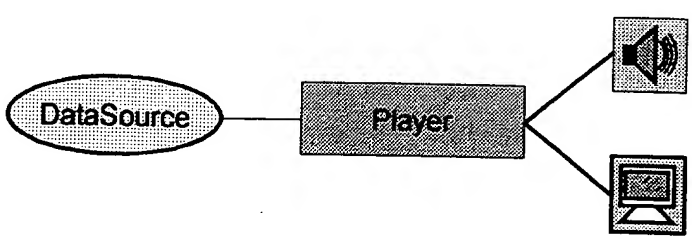


Figure 2-10: JMF player model.

A `Player` does not provide any control over the processing that it performs or how it renders the media data.

`Player` supports standardized user control and relaxes some of the operational restrictions imposed by `Clock` and `Controller`.

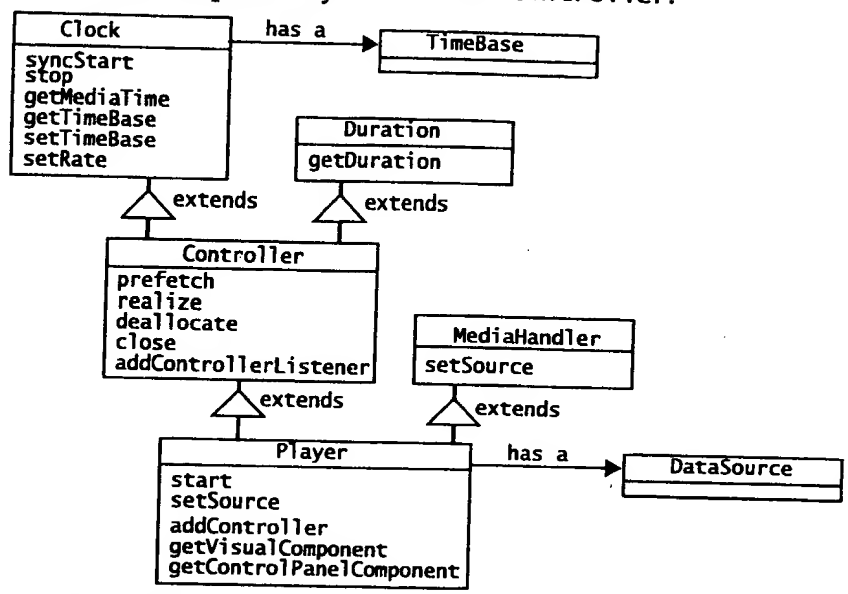


Figure 2-11: JMF players.

Player States

A `Player` can be in one of six states. The `Clock` interface defines the two primary states: *Stopped* and *Started*. To facilitate resource management, `Controller` breaks the *Stopped* state down into five standby states: *Unrealized, Realizing, Realized, Prefetching,* and *Prefetched*.

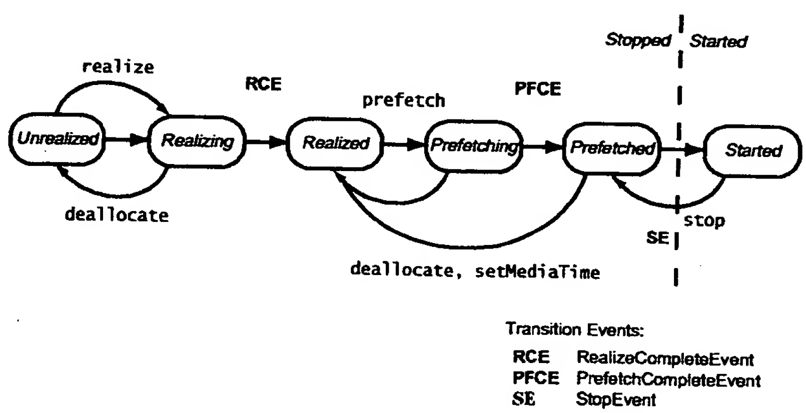


Figure 2-12: Player states.

In normal operation, a `Player` steps through each state until it reaches the *Started* state:

- A `Player` in the *Unrealized* state has been instantiated, but does not yet know anything about its media. When a media `Player` is first created, it is *Unrealized*.
- When `realize` is called, a `Player` moves from the *Unrealized* state into the *Realizing* state. A *Realizing* `Player` is in the process of determining its resource requirements. During realization, a `Player` acquires the resources that it only needs to acquire once. These might include rendering resources other than exclusive-use resources. (Exclusive-use resources are limited resources such as particular hardware devices that can only be used by one `Player` at a time; such resources are acquired during *Prefetching*.) A *Realizing* `Player` often downloads assets over the network.
- When a `Player` finishes *Realizing*, it moves into the *Realized* state. A *Realized* `Player` knows what resources it needs and information about the type of media it is to present. Because a *Realized* `Player` knows how to render its data, it can provide visual components and controls. Its connections to other objects in the system are in place, but it does not own any resources that would prevent another `Player` from starting.
- When `prefetch` is called, a `Player` moves from the *Realized* state into the *Prefetching* state. A *Prefetching* `Player` is preparing to present its media. During this phase, the `Player` preloads its media data, obtains exclusive-use resources, and does whatever else it needs to do to prepare itself to play. *Prefetching* might have to recur if a `Player` object's media presentation is repositioned, or if a change in the `Player` object's rate requires that additional buffers be acquired or alternate processing take place.
- When a `Player` finishes *Prefetching*, it moves into the *Prefetched* state. A *Prefetched* `Player` is ready to be started.
- Calling `start` puts a `Player` into the *Started* state. A *Started* `Player` object's time-base time and media time are mapped and its clock is running, though the `Player` might be waiting for a particular time to begin presenting its media data.

A `Player` posts `TransitionEvents` as it moves from one state to another. The `ControllerListener` interface provides a way for your program to determine what state a `Player` is in and to respond appropriately. For example, when your program calls an asynchronous method on a `Player`

or Processor, it needs to listen for the appropriate event to determine when the operation is complete.

Using this event reporting mechanism, you can manage a Player object's start latency by controlling when it begins *Realizing* and *Prefetching*. It also enables you to determine whether or not the Player is in an appropriate state before calling methods on the Player.

Methods Available in Each Player State

To prevent race conditions, not all methods can be called on a Player in every state. The following table identifies the restrictions imposed by JMF. If you call a method that is illegal in a Player object's current state, the Player throws an error or exception.

Method	Unrealized Player	Realized Player	Prefetched Player	Started Player
addController	NotRealizedError	legal	legal	ClockStartedError
deallocate	legal	legal	legal	ClockStartedError
getControlPanelComponent	NotRealizedError	legal	legal	legal
getGainControl	NotRealizedError	legal	legal	legal
getStartLatency	NotRealizedError	legal	legal	legal
getTimeBase	NotRealizedError	legal	legal	legal
getVisualComponent	NotRealizedError	legal	legal	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	legal
removeController	NotRealizedError	legal	legal	ClockStartedError
setMediaTime	NotRealizedError	legal	legal	legal
setRate	NotRealizedError	legal	legal	legal
setStopTime	NotRealizedError	legal	legal	StopTimeSetError if previously set
setTimeBase	NotRealizedError	legal	legal	ClockStartedError
syncStart	NotPrefetchedError	NotPrefetchedError	legal	ClockStartedError

Table 2-1: Method restrictions for players.

Processors

Processors can also be used to present media data. A Processor is just a specialized type of Player that provides control over what processing is performed on the input media stream. A Processor supports all of the same presentation controls as a Player.

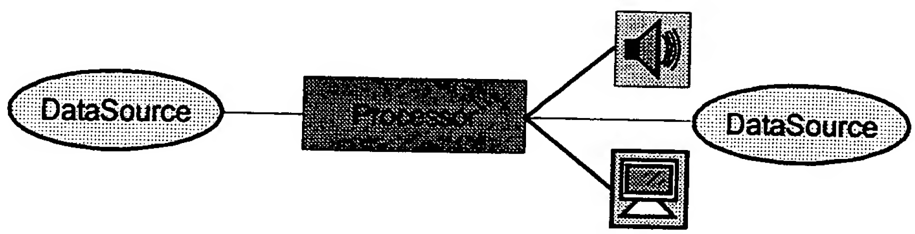


Figure 2-13: JMF processor model.

In addition to rendering media data to presentation devices, a Processor can output media data through a DataSource so that it can be presented by another Player or Processor, further manipulated by another Processor, or delivered to some other destination, such as a file.

For more information about Processors, see "Processing" on page 32.

Presentation Controls

In addition to the standard presentation controls defined by Controller, a Player or Processor might also provide a way to adjust the playback volume. If so, you can retrieve its GainControl by calling getGainControl. A GainControl object posts a GainChangeEvent whenever the gain is modified. By implementing the GainChangeListener interface, you can respond to gain changes. For example, you might want to update a custom gain control Component.

Additional custom Control types might be supported by a particular Player or Processor implementation to provide other control behaviors and expose custom user interface components. You access these controls through the getControls method.

For example, the CachingControl interface extends Control to provide a mechanism for displaying a download progress bar. If a Player can report its download progress, it implements this interface. To find out if a Player supports CachingControl, you can call getControl(CachingControl) or use getControls to get a list of all the supported Controls.

Standard User Interface Components

A `Player` or `Processor` generally provides two standard user interface components, a visual component and a control-panel component.You can access these `Components` directly through the `getVisualComponent` and `getControlPanelComponent` methods.

You can also implement custom user interface components, and use the event listener mechanism to determine when they need to be updated.

Controller Events

The `ControllerEvents` posted by a `Controller` such as a `Player` or `Processor` fall into three categories: change notifications, closed events, and transition events:

- Change notification events such as `RateChangeEvent`, `DurationUpdateEvent`, and `FormatChangeEvent` indicate that some attribute of the `Controller` has changed, often in response to a method call. For example, a `Player` posts a `RateChangeEvent` when its rate is changed by a call to `setRate`.
- `TransitionEvents` allow your program to respond to changes in a `Controller` object's state. A `Player` posts transition events whenever it moves from one state to another. (See "Player States" on page 26 for more information about the states and transitions.)
- `ControllerClosedEvents` are posted by a `Controller` when it shuts down. When a `Controller` posts a `ControllerClosedEvent`, it is no longer usable. A `ControllerErrorEvent` is a special case of `ControllerClosedEvent`. You can listen for `ControllerErrorEvents` so that your program can respond to `Controller` malfunctions to minimize the impact on the user.

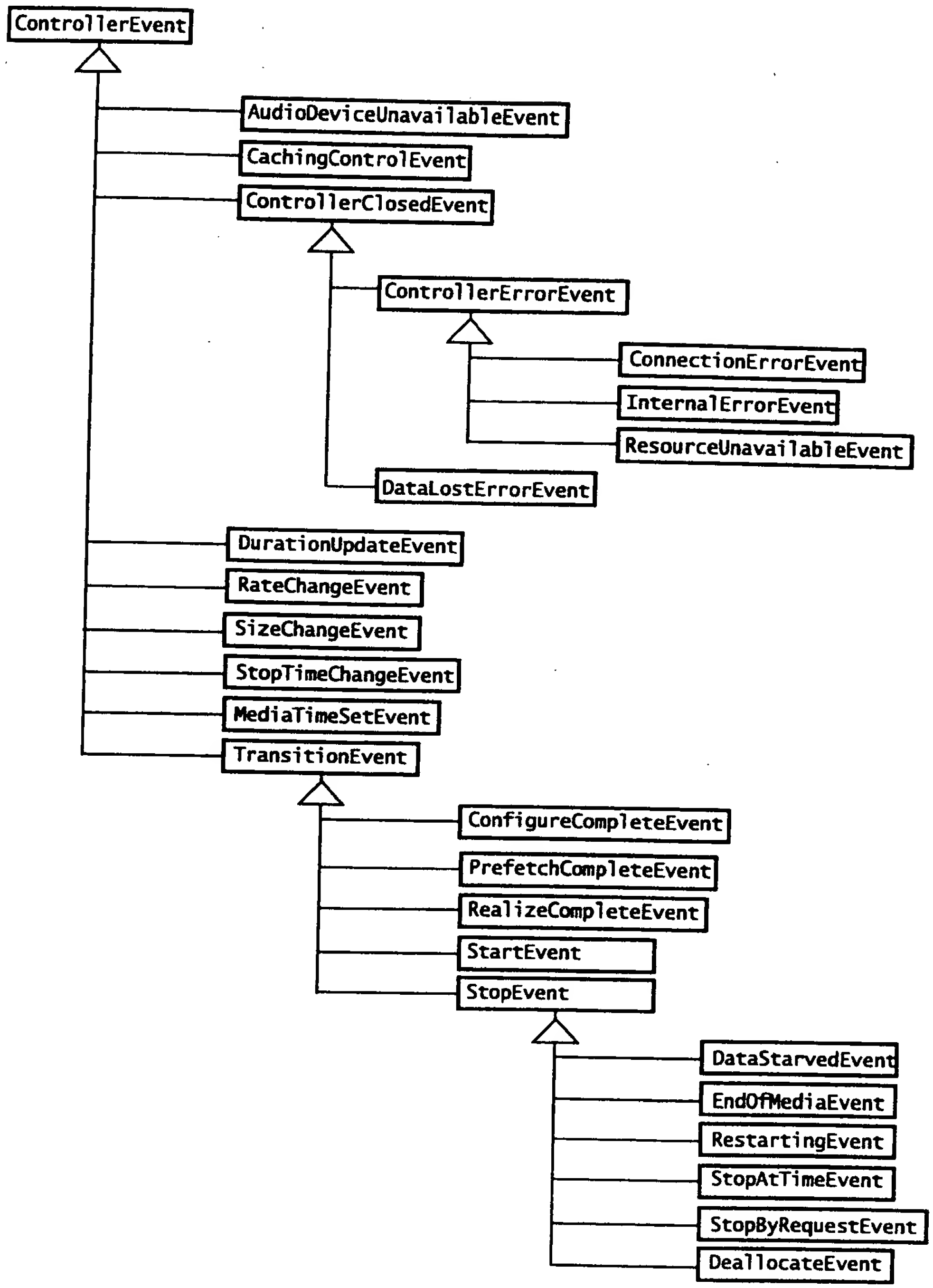


Figure 2-14: JMF events.

Processing

A `Processor` is a `Player` that takes a `DataSource` as input, performs some user-defined processing on the media data, and then outputs the processed media data.

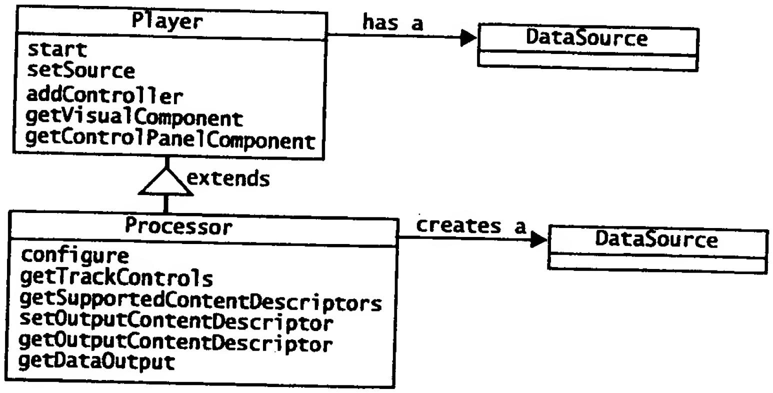


Figure 2-15: JMF processors.

A `Processor` can send the output data to a presentation device or to a `DataSource`. If the data is sent to a `DataSource`, that `DataSource` can be used as the input to another `Player` or `Processor`, or as the input to a `DataSink`.

While the processing performed by a `Player` is predefined by the implementor, a `Processor` allows the application developer to define the type of processing that is applied to the media data. This enables the application of effects, mixing, and compositing in real-time.

The processing of the media data is split into several stages:

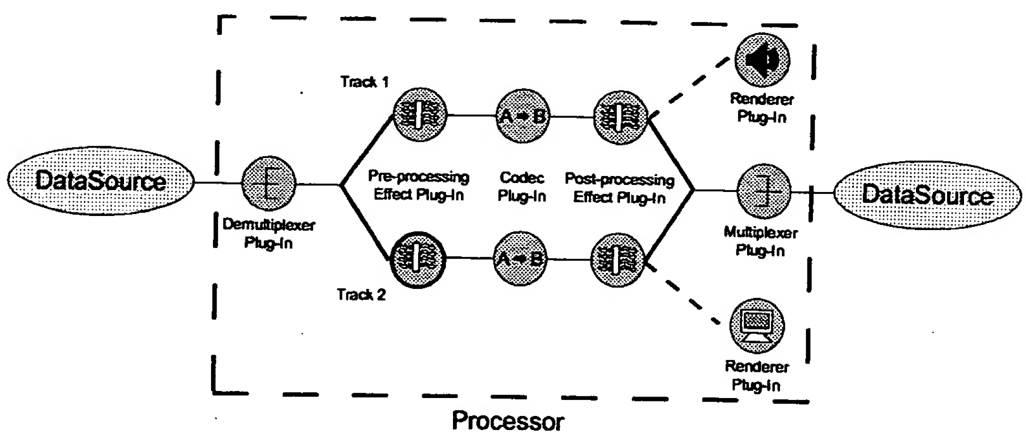


Figure 2-16: Processor stages.

- Demultiplexing is the process of parsing the input stream. If the stream contains multiple tracks, they are extracted and output

separately. For example, a QuickTime file might be demultiplexed into separate audio and video tracks. Demultiplexing is performed automatically whenever the input stream contains multiplexed data.

- Pre-Processing is the process of applying effect algorithms to the tracks extracted from the input stream.
- Transcoding is the process of converting each track of media data from one input format to another. When a data stream is converted from a compressed type to an uncompressed type, it is generally referred to as decoding. Conversely, converting from an uncompressed type to a compressed type is referred to as encoding.
- Post-Processing is the process of applying effect algorithms to decoded tracks.
- Multiplexing is the process of interleaving the transcoded media tracks into a single output stream. For example, separate audio and video tracks might be multiplexed into a single MPEG-1 data stream. You can specify the data type of the output stream with the `Processor` `setOutputContentDescriptor` method.
- Rendering is the process of presenting the media to the user.

The processing at each stage is performed by a separate processing component. These processing components are JMF *plug-ins*. If the `Processor` supports `TrackControls`, you can select which plug-ins you want to use to process a particular track. There are five types of JMF plug-ins:

- `Demultiplexer`—parses media streams such as WAV, MPEG or QuickTime. If the stream is multiplexed, the separate tracks are extracted.
- `Effect`—performs special effects processing on a track of media data.
- `Codec`—performs data encoding and decoding.
- `Multiplexer`—combines multiple tracks of input data into a single interleaved output stream and delivers the resulting stream as an output `DataSource`.
- `Renderer`—processes the media data in a track and delivers it to a destination such as a screen or speaker.

Processor States

A `Processor` has two additional standby states, *Configuring* and *Configured*, which occur before the `Processor` enters the *Realizing* state..

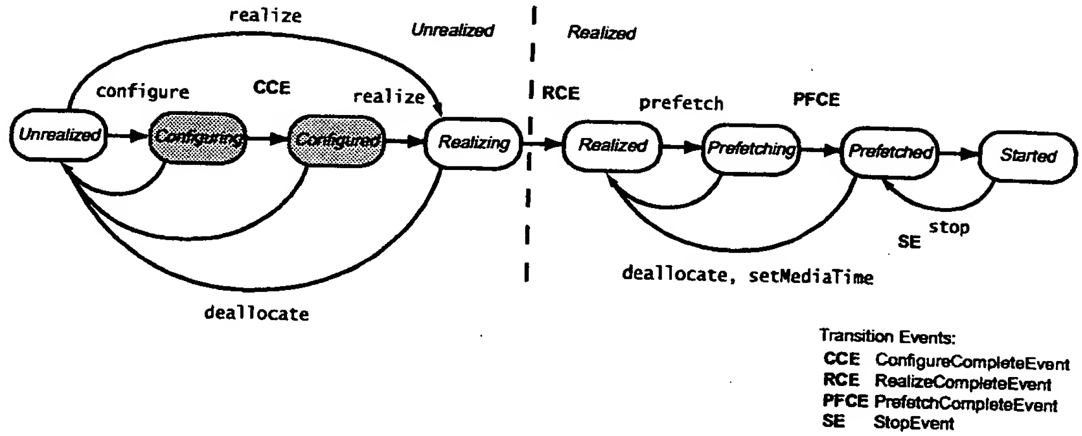


Figure 2-17: Processor states.

- A `Processor` enters the *Configuring* state when `configure` is called. While the `Processor` is in the *Configuring* state, it connects to the `DataSource`, demultiplexes the input stream, and accesses information about the format of the input data.
- The `Processor` moves into the *Configured* state when it is connected to the `DataSource` and data format has been determined. When the `Processor` reaches the *Configured* state, a `ConfigureCompleteEvent` is posted.
- When `Realize` is called, the `Processor` is transitioned to the `Realized` state. Once the `Processor` is *Realized* it is fully constructed.

While a `Processor` is in the *Configured* state, `getTrackControls` can be called to get the `TrackControl` objects for the individual tracks in the media stream. These `TrackControl` objects enable you specify the media processing operations that you want the `Processor` to perform.

Calling `realize` directly on an *Unrealized* `Processor` automatically transitions it through the *Configuring* and *Configured* states to the *Realized* state. When you do this, you cannot configure the processing options through the `TrackControls`—the default `Processor` settings are used.

Calls to the `TrackControl` methods once the `Processor` is in the *Realized* state will typically fail, though some `Processor` implementations might support them.

Methods Available in Each Processor State

Since a `Processor` is a type of `Player`, the restrictions on when methods can be called on a `Player` also apply to `Processors`. Some of the `Processor`-specific methods also are restricted to particular states. The following table shows the restrictions that apply to a `Processor`. If you call a method that is illegal in the current state, the `Processor` throws an error or exception.

Method	Unrealized Processor	Configuring Processor	Configured Processor	Realized Processor
addController	NotRealizedError	NotRealizedError	NotRealizedError	legal
deallocate	legal	legal	legal	legal
getControlPanelComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
getControls	legal	legal	legal	legal
getDataOutput	NotRealizedError	NotRealizedError	NotRealizedError	legal
getGainControl	NotRealizedError	NotRealizedError	NotRealizedError	legal
getOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	legal
getStartLatency	NotRealizedError	NotRealizedError	NotRealizedError	legal
getSupportedContent-Descriptors	legal	legal	legal	legal
getTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
getTrackControls	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
getVisualComponent	NotRealizedError	NotRealizedError	NotRealizedError	legal
mapToTimeBase	ClockStoppedException	ClockStoppedException	ClockStoppedException	ClockStopped-Exception
realize	legal	legal	legal	legal
removeController	NotRealizedError	NotRealizedError	NotRealizedError	legal
setOutputContentDescriptor	NotConfiguredError	NotConfiguredError	legal	FormatChange-Exception
setMediaTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setRate	NotRealizedError	NotRealizedError	NotRealizedError	legal
setStopTime	NotRealizedError	NotRealizedError	NotRealizedError	legal
setTimeBase	NotRealizedError	NotRealizedError	NotRealizedError	legal
syncStart	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError	NotPrefetchedError

Table 2-2: Method restrictions for processors.

Processing Controls

You can control what processing operations the Processor performs on a track through the TrackControl for that track. You call Processor getTrackControls to get the TrackControl objects for all of the tracks in the media stream.

Through a TrackControl, you can explicitly select the Effect, Codec, and Renderer plug-ins you want to use for the track. To find out what options are available, you can query the PlugInManager to find out what plug-ins are installed.

To control the transcoding that's performed on a track by a particular Codec, you can get the Controls associated with the track by calling the TrackControl getControls method. This method returns the codec controls available for the track, such as BitRateControl and QualityControl. (For more information about the codec controls defined by JMF, see "Controls" on page 20.)

If you know the output data format that you want, you can use the setFormat method to specify the Format and let the Processor choose an appropriate codec and renderer. Alternatively, you can specify the output format when the Processor is created by using a ProcessorModel. A ProcessorModel defines the input and output requirements for a Processor. When a ProcessorModel is passed to the appropriate Manager create method, the Manager does its best to create a Processor that meets the specified requirements.

Data Output

The getDataOutput method returns a Processor object's output as a DataSource. This DataSource can be used as the input to another Player or Processor or as the input to a *data sink*. (For more information about data sinks, see "Media Data Storage and Transmission" on page 37.)

A Processor object's output DataSource can be of any type: PushDataSource, PushBufferDataSource, PullDataSource, or PullBufferDataSource.

Not all Processor objects output data—a Processor can render the processed data instead of outputting the data to a DataSource. A Processor that renders the media data is essentially a configurable Player.

Capture

A multimedia capturing device can act as a source for multimedia data delivery. For example, a microphone can capture raw audio input or a digital video capture board might deliver digital video from a camera. Such capture devices are abstracted as `DataSources`. For example, a device that provides timely delivery of data can be represented as a `PushDataSource`. Any type of `DataSource` can be used as a capture `DataSource`: `PushDataSource`, `PushBufferDataSource`, `PullDataSource`, or `PullBufferDataSource`.

Some devices deliver multiple data streams—for example, an audio/video conferencing board might deliver both an audio and a video stream. The corresponding `DataSource` can contain multiple `SourceStreams` that map to the data streams provided by the device.

Media Data Storage and Transmission

A `DataSink` is used to read media data from a `DataSource` and render the media to some destination—generally a destination other than a presentation device. A particular `DataSink` might write data to a file, write data across the network, or function as an RTP broadcaster. (For more information about using a `DataSink` as an RTP broadcaster, see "Transmitting RTP Data With a Data Sink" on page 149.)

Like `Players`, `DataSink` objects are constructed through the `Manager` using a `DataSource`. A `DataSink` can use a `StreamWriterControl` to provide additional control over how data is written to a file. See "Writing Media Data to a File" on page 74 for more information about how `DataSink` objects are used.

Storage Controls

A `DataSink` posts a `DataSinkEvent` to report on its status. A `DataSinkEvent` can be posted with a reason code, or the `DataSink` can post one of the following `DataSinkEvent` subtypes:

- `DataSinkErrorEvent`, which indicates that an error occurred while the `DataSink` was writing data.
- `EndOfStreamEvent`, which indicates that the entire stream has successfully been written.

To respond to events posted by a DataSink, you implement the DataSinkListener interface.

Extensibility

You can extend JMF by implementing custom plug-ins, media handlers, and data sources.

Implementing Plug-Ins

By implementing one of the JMF plug-in interfaces, you can directly access and manipulate the media data associated with a Processor:

- Implementing the Demultiplexer interface enables you to control how individual tracks are extracted from a multiplexed media stream.
- Implementing the Codec interface enables you to perform the processing required to decode compressed media data, convert media data from one format to another, and encode raw media data into a compressed format.
- Implementing the Effect interface enables you to perform custom processing on the media data.
- Implementing the Multiplexer interface enables you to specify how individual tracks are combined to form a single interleaved output stream for a Processor.
- Implementing the Renderer interface enables you to control how data is processed and rendered to an output device.

Note: The JMF Plug-In API is part of the official JMF API, but JMF Players and Processors are not required to support plug-ins. Plug-ins won't work with JMF 1.0-based Players and some Processor implementations might choose not to support them. The reference implementation of JMF 2.0 provided by Sun Microsystems, Inc. and IBM Corporation fully supports the plug-in API.

Custom Codec, Effect, and Renderer plug-ins are available to a Processor through the TrackControl interface. To make a plug-in available to a default Processor or a Processor created with a ProcessorModel, you need to register it with the PlugInManager. Once you've registered your plug-in, it is included in the list of plug-ins returned by the PlugInManager get-

`PlugInList` method and can be accessed by the `Manager` when it constructs a `Processor` object.

Implementing MediaHandlers and DataSources

If the JMF Plug-In API doesn't provide the degree of flexibility that you need, you can directly implement several of the key JMF interfaces: `Controller`, `Player`, `Processor`, `DataSource`, and `DataSink`. For example, you might want to implement a high-performance `Player` that is optimized to present a single media format or a `Controller` that manages a completely different type of time-based media.

The `Manager` mechanism used to construct `Player`, `Processor`, `DataSource`, and `DataSink` objects enables custom implementations of these JMF interfaces to be used seamlessly with JMF. When one of the `create` methods is called, the `Manager` uses a well-defined mechanism to locate and construct the requested object. Your custom class can be selected and constructed through this mechanism once you register a unique package prefix with the `PackageManager` and put your class in the appropriate place in the predefined package hierarchy.

MediaHandler Construction

`Players`, `Processors`, and `DataSinks` are all types of `MediaHandlers`—they all read data from a `DataSource`. A `MediaHandler` is always constructed for a particular `DataSource`, which can be either identified explicitly or with a `MediaLocator`. When one of the `create`*`MediaHandler`* methods is called, `Manager` uses the content-type name obtained from the `DataSource` to find and create an appropriate `MediaHandler` object.

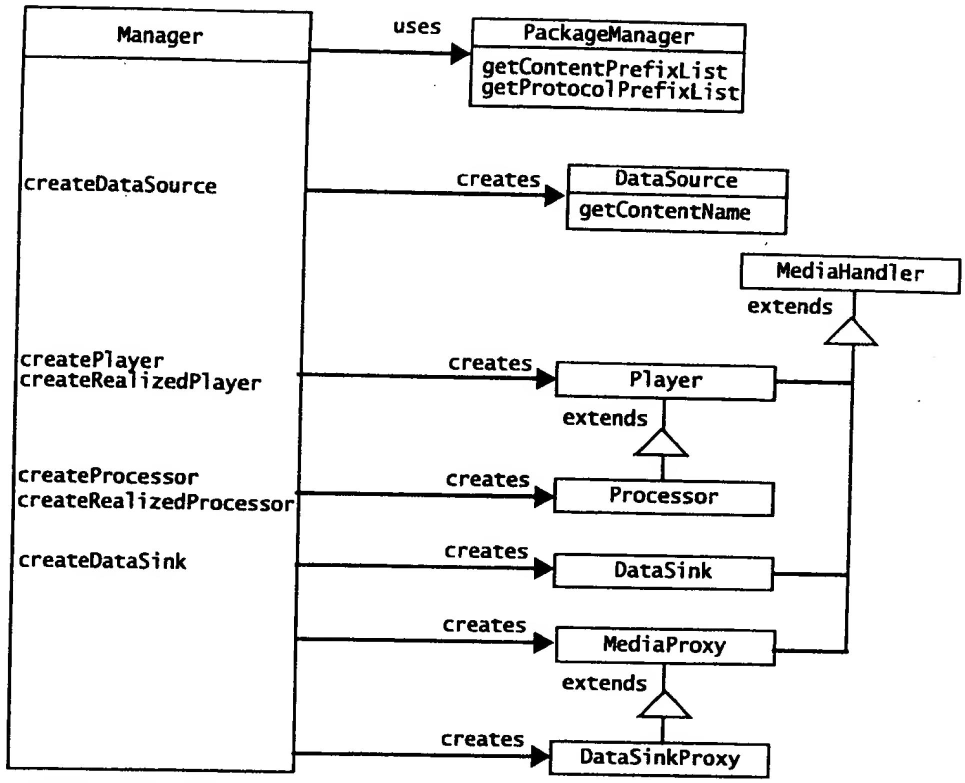


Figure 2-18: JMF media handlers.

JMF also supports another type of `MediaHandler`, `MediaProxy`. A `MediaProxy` processes content from one `DataSource` to create another. Typically, a `MediaProxy` reads a text configuration file that contains all of the information needed to make a connection to a server and obtain media data. To create a `Player` from a `MediaProxy`, `Manager`:

1. Constructs a `DataSource` for the protocol described by the `MediaLocator`
2. Uses the content-type of the `DataSource` to construct a `MediaProxy` to read the configuration file.
3. Gets a new `DataSource` from the `MediaProxy`.
4. Uses the content-type of the new `DataSource` to construct a `Player`.

The mechanism that `Manager` uses to locate and instantiate an appropriate `MediaHandler` for a particular `DataSource` is basically the same for all types of `MediaHandlers`:

- Using the list of installed content package-prefixes retrieved from `PackageManager`, `Manager` generates a search list of available `MediaHandler` classes.
- `Manager` steps through each class in the search list until it finds a class named `Handler` that can be constructed and to which it can attach the `DataSource`.

When constructing `Players` and `Processors`, `Manager` generates the search list of available handler classes from the list of installed *content package-prefixes* and the content-type name of the `DataSource`. To search for `Players`, `Manager` looks for classes of the form:

```
<content package-prefix>.media.content.<content-type>.Handler
```

To search for `Processors`, `Manager` looks for classes of the form:

```
<content package-prefix>.media.processor.<content-type>.Handler
```

If the located `MediaHandler` is a `MediaProxy`, `Manager` gets a new `DataSource` from the `MediaProxy` and repeats the search process.

If no appropriate `MediaHandler` can be found, the search process is repeated, substituting `unknown` for the content-type name. The `unknown` content type is supported by generic `Players` that are capable of handling a large variety of media types, often in a platform-dependent way.

Because a `DataSink` renders the data it reads from its `DataSource` to an output destination, when a `DataSink` is created the destination must also be taken into account. When constructing `DataSinks`, `Manager` uses the list of content package-prefixes and the protocol from the `MediaLocator` that identifies the destination. For each content package-prefix, `Manager` adds to the search list a class name of the form:

```
<content package-prefix>.media.datasink.protocol.Handler
```

If the located `MediaHandler` is a `DataSink`, `Manager` instantiates it, sets its `DataSource` and `MediaLocator`, and returns the resulting `DataSink` object. If the handler is a `DataSinkProxy`, `Manager` retrieves the content type of the proxy and generates a list of `DataSink` classes that support the protocol of the destination `Medialocator` and the content type returned by the proxy:

```
<content package-prefix>.media.datasink.protocol.<content-type>.Handler
```

The process continues until an appropriate `DataSink` is located or the `Manager` has iterated through all of the content package-prefixes.

DataSource Construction

`Manager` uses the same mechanism to construct `DataSources` that it uses to construct `MediaHandlers`, except that it generates the search list of `DataSource` class names from the list of installed *protocol package-prefixes*.

For each protocol package-prefix, `Manager` adds to the search list a class name of the form:

```
<protocol package-prefix>.media.protocol.<protocol>.DataSource
```

`Manager` steps through each class in the list until it finds a `DataSource` that it can instantiate and to which it can attach the `MediaLocator`.

3

Presenting Time-Based Media with JMF

To present time-based media such as audio or video with JMF, you use a Player. Playback can be controlled programmatically, or you can display a control-panel component that enables the user to control playback interactively. If you have several media streams that you want to play, you need to use a separate Player for each one. to play them in sync, you can use one of the Player objects to control the operation of the others.

A Processor is a special type of Player that can provide control over how the media data is processed before it is presented. Whether you're using a basic Player or a more advanced Processor to present media content, you use the same methods to manage playback. For information about how to control what processing is performed by a Processor, see "Processing Time-Based Media with JMF" on page 71.

The MediaPlayer bean is a Java Bean that encapsulates a JMF player to provide an easy way to present media from an applet or application. The MediaPlayer bean automatically constructs a new Player when a different media stream is selected, which makes it easier to play a series of media clips or allow the user to select which media clip to play. For information about using the MediaPlayer bean, see "Presenting Media with the Media-Player Bean" on page 66

Controlling a Player

To play a media stream, you need to construct a Player for the stream, configure the Player and prepare it to run, and then start the Player to begin playback.

Creating a Player

You create a Player indirectly through the media Manager. To display the Player, you get the Player object's components and add them to your applet's presentation space or application window.

When you need to create a new Player, you request it from the Manager by calling createPlayer or createProcessor. The Manager uses the media URL or MediaLocator that you specify to create an appropriate Player. A URL can only be successfully constructed if the appropriate corresponding URL-StreamHandler is installed. MediaLocator doesn't have this restriction.

Blocking Until a Player is Realized

Many of the methods that can be called on a Player require the Player to be in the *Realized* state. One way to guarantee that a Player is *Realized* when you call these methods is to use the Manager createRealizedPlayer method to construct the Player. This method provides a convenient way to create and realize a Player in a single step. When this method is called, it blocks until the Player is *Realized*. Manager provides an equivalent createRealizeProcessor method for constructing a *Realized* Processor.

Note: Be aware that blocking until a Player or Processor is *Realized* can produce unsatisfactory results. For example, if createRealizedPlayer is called in an applet, Applet.start and Applet.stop will not be able to interrupt the construction process.

Using a ProcessorModel to Create a Processor

A Processor can also be created using a ProcessorModel. The ProcessorModel defines the input and output requirements for the Processor and the Manager does its best to create a Processor that meets these requirements. To create a Processor using a ProcessorModel, you call the Manager.createRealizedProcessor method. Example 3-1 creates a *Realized* Processor that can produce IMA4-encoded stereo audio tracks with a 44.1 kHz sample rate and a 16-bit sample size.

Example 3-1: Constructing a Processor with a ProcessorModel.

```
AudioFormat afs[] = new AudioFormat[1];
afs[0] = new AudioFormat("ima4", 44100, 16, 2);
Manager.createRealizedProcessor(new ProcessorModel(afs, null));
```

Since the `ProcessorModel` does not specify a source URL in this example, `Manager` implicitly finds a capture device that can capture audio and then creates a `Processor` that can encode that into IMA4.

Note that when you create a *Realized* `Processor` with a `ProcessorModel` you will not be able to specify processing options through the `Processor` object's `TrackControls`. For more information about specifying processing options for a Processor, see "Processing Time-Based Media with JMF" on page 71.

Displaying Media Interface Components

A `Player` generally has two types of user interface components, a visual component and a control-panel component. Some `Player` implementations can display additional components, such as volume controls and download-progress bars.

Displaying a Visual Component

A visual component is where a `Player` presents the visual representation of its media, if it has one. Even an audio `Player` might have a visual component, such as a waveform display or animated character.

To display a `Player` object's visual component, you:

1. Get the component by calling `getVisualComponent`.
2. Add it to the applet's presentation space or application window.

You can access the `Player` object's display properties, such as its *x* and *y* coordinates, through its visual component. The layout of the `Player` components is controlled through the AWT layout manager.

Displaying a Control Panel Component

A `Player` often has a control panel that allows the user to control the media presentation. For example, a `Player` might be associated with a set of buttons to start, stop, and pause the media stream, and with a slider control to adjust the volume.

Every `Player` provides a default control panel. To display the default control panel:

1. Call `getControlPanelComponent` to get the `Component`.
2. Add the returned `Component` to your applet's presentation space or application window.

If you prefer to define a custom user-interface, you can implement custom GUI `Components` and call the appropriate `Player` methods in response to user actions. If you register the custom components as `ControllerListeners`, you can also update them when the state of the `Player` changes.

Displaying a Gain-Control Component

`Player` implementations that support audio gain adjustments implement the `GainControl` interface. `GainControl` provides methods for adjusting the audio volume, such as `setLevel` and `setMute`. To display a `GainControl` `Component` if the `Player` provides one, you:

1. Call `getGainControl` to get the `GainControl` from the `Player`. If the `Player` returns null, it does not support the `GainControl` interface.
2. Call `getControlComponent` on the returned `GainControl`.
3. Add the returned `Component` to your applet's presentation space or application window.

Note that `getControls` does not return a `Player` object's `GainControl`. You can only access the `GainControl` by calling `getGainControl`.

Displaying Custom Control Components

Many `Players` have other properties that can be managed by the user. For example, a video `Player` might allow the user to adjust brightness and contrast, which are not managed through the `Player` interface. You can find out what custom controls a `Player` supports by calling the `getControls` method.

For example, you can call `getControls` to determine if a `Player` supports the `CachingControl` interface.

Example 3-2: Using `getControls` to find out what `Controls` are supported.

```
Control[] controls = player.getControls();
    for (int i = 0; i < controls.length; i++) {
        if (controls[i] instanceof CachingControl) {
            cachingControl = (CachingControl) controls[i];
        }
    }
```

Displaying a Download-Progress Component

The `CachingControl` interface is a special type of `Control` implemented by `Players` that can report their download progress. A `CachingControl` provides a default progress-bar component that is automatically updated as the download progresses. To use the default progress bar in an applet:

1. Implement the `ControllerListener` interface and listen for `CachingControlEvents` in `controllerUpdate`.
2. The first time you receive a `CachingControlEvent`:
   a. Call `getCachingControl` on the event to get the caching control.
   b. Call `getProgressBar` on the `CachingControl` to get the default progress bar component.
   c. Add the progress bar component to your applet's presentation space.
3. Each time you receive a `CachingControlEvent`, check to see if the download is complete. When `getContentProgress` returns the same value as `getContentLength`, remove the progress bar.

The `Player` posts a `CachingControlEvent` whenever the progress bar needs to be updated. If you implement your own progress bar component, you can listen for this event and update the download progress whenever `CachingControlEvent` is posted.

Setting the Playback Rate

The `Player` object's rate determines how media time changes with respect to time-base time; it defines how many units a `Player` object's media time advances for every unit of time-base time. The `Player` object's rate can be thought of as a temporal scale factor. For example, a rate of 2.0 indicates

that media time passes twice as fast as the time-base time when the `Player` is started.

In theory, a `Player` object's rate could be set to any real number, with negative rates interpreted as playing the media in reverse. However, some media formats have dependencies between frames that make it impossible or impractical to play them in reverse or at non-standard rates.

To set the rate, you call `setRate` and pass in the temporal scale factor as a float value. When `setRate` is called, the method returns the rate that is actually set, even if it has not changed. `Players` are only guaranteed to support a rate of 1.0.

Setting the Start Position

Setting a `Player` object's media time is equivalent to setting a read position within a media stream. For a media data source such as a file, the media time is bounded; the maximum media time is defined by the end of the media stream.

To set the media time you call `setMediaTime` and pass in a `Time` object that represents the time you want to set.

Frame Positioning

Some `Players` allow you to seek to a particular frame of a video. This enables you to easily set the start position to the beginning of particular frame without having to specify the exact media time that corresponds to that position. `Players` that support frame positioning implement the `FramePositioningControl`.

To set the frame position, you call the `FramePositioningControl seek` method. When you seek to a frame, the `Player` object's media time is set to the value that corresponds to the beginning of that frame and a `MediaTimeSetEvent` is posted.

Some `Players` can convert between media times and frame positions. You can use the `FramePositioningControl mapFrameToTime` and `mapTimeToFrame` methods to access this information, if it's available. (`Players` that support `FramePositioningControl` are not required to export this information.) Note that there is not a one-to-one correspondence between media times and frames —a frame has a duration, so several different media times might map to the same frame. (See "Getting the Media Time" on page 53 for more information.)

Preparing to Start

Most media `Players` cannot be started instantly. Before the `Player` can start, certain hardware and software conditions must be met. For example, if the `Player` has never been started, it might be necessary to allocate buffers in memory to store the media data. Or, if the media data resides on a network device, the `Player` might have to establish a network connection before it can download the data. Even if the `Player` has been started before, the buffers might contain data that is not valid for the current media position.

Realizing and Prefetching a Player

JMF breaks the process of preparing a `Player` to start into two phases, *Realizing* and *Prefetching*. *Realizing* and *Prefetching* a `Player` before you start it minimizes the time it takes the `Player` to begin presenting media when `start` is called and helps create a highly-responsive interactive experience for the user. Implementing the `ControllerListener` interface allows you to control when these operations occur.

Note: `Processor` introduces a third phase to the preparation process called *Configuring*. During this phase, `Processor` options can be selected to control how the `Processor` manipulates the media data. For more information, see "Selecting Track Processing Options" on page 72.

You call `realize` to move the `Player` into the *Realizing* state and begin the realization process. You call `prefetch` to move the `Player` into the *Prefetching* state and initiate the prefetching process. The `realize` and `prefetch` methods are asynchronous and return immediately. When the `Player` completes the requested operation, it posts a `RealizeCompleteEvent` or `PrefetchCompleteEvent`. "Player States" on page 26 describes the operations that a `Player` performs in each of these states.

A `Player` in the *Prefetched* state is prepared to start and its start-up latency cannot be further reduced. However, setting the media time through `setMediaTime` might return the `Player` to the *Realized* state and increase its start-up latency.

Keep in mind that a *Prefetched* `Player` ties up system resources. Because some resources, such as sound cards, might only be usable by one program at a time, having a `Player` in the *Prefetched* state might prevent other `Players` from starting.

Determining the Start Latency

To determine how much time is required to start a `Player`, you can call `getStartLatency`. For `Players` that have a variable start latency, the return value of `getStartLatency` represents the maximum possible start latency. For some media types, `getStartLatency` might return `LATENCY_UNKNOWN`.

The start-up latency reported by `getStartLatency` might differ depending on the `Player` object's current state. For example, after a `prefetch` operation, the value returned by `getStartLatency` is typically smaller. A `Controller` that can be added to a `Player` will return a useful value once it is *Prefetched*. (For more information, see "Using a Player to Synchronize Controllers" on page 57.)

Starting and Stopping the Presentation

The `Clock` and `Player` interfaces define the methods for starting and stopping presentation.

Starting the Presentation

You typically start the presentation of media data by calling `start`. The `start` method tells the `Player` to begin presenting media data as soon as possible. If necessary, `start` prepares the `Player` to start by performing the realize and prefetch operations. If `start` is called on a *Started* `Player`, the only effect is that a `StartEvent` is posted in acknowledgment of the method call.

Clock defines a `syncStart` method that can be used for synchronization. See "Synchronizing Multiple Media Streams" on page 56 for more information.

To start a `Player` at a specific point in a media stream:

1. Specify the point in the media stream at which you want to start by calling `setMediaTime`.
2. Call `start` on the `Player`.

Stopping the Presentation

There are four situations in which the presentation will stop:

- When the `stop` method is called

- When the specified stop time is reached
- When there's no more media data to present
- When the media data is being received too slowly for acceptable playback

When a `Player` is stopped, its media time is frozen if the source of the media can be controlled. If the `Player` is presenting streamed media, it might not be possible to freeze the media time. In this case, only the receipt of the media data is stopped—the data continues to be streamed and the media time continues to advance.

When a *Stopped* `Player` is restarted, if the media time was frozen, presentation resumes from the stop time. If media time could not be frozen when the `Player` was stopped, reception of the stream resumes and playback begins with the newly-received data.

To stop a `Player` immediately, you call the `stop` method. If you call `stop` on a *Stopped* `Player`, the only effect is that a `StopByRequestEvent` is posted in acknowledgment of the method call.

Stopping the Presentation at a Specified Time

You can call `setStopTime` to indicate when a `Player` should stop. The `Player` stops when its media time passes the specified stop time. If the `Player` object's rate is positive, the `Player` stops when the media time becomes greater than or equal to the stop time. If the `Player` object's rate is negative, the `Player` stops when the media time becomes less than or equal to the stop time. The `Player` stops immediately if its current media time is already beyond the specified stop time.

For example, assume that a `Player` object's media time is 5.0 and `setStopTime` is called to set the stop time to 6.0. If the `Player` object's rate is positive, media time is increasing and the `Player` will stop when the media time becomes greater than or equal to 6.0. However, if the `Player` object's rate is negative, it is playing in reverse and the `Player` will stop immediately because the media time is already beyond the stop time. (For more information about `Player` rates, see "Setting the Playback Rate" on page 47.)

You can always call `setStopTime` on a *Stopped* `Player`. However, you can only set the stop time on a *Started* `Player` if the stop time is not currently

set. If the *Started* `Player` already has a stop time, `setStopTime` throws an error.

You can call `getStopTime` to get the currently scheduled stop time. If the clock has no scheduled stop time, `getStopTime` returns `Clock.RESET`. To remove the stop time so that the `Player` continues until it reaches end-of-media, call `setStopTime(Clock.RESET)`.

Releasing Player Resources

The `deallocate` method tells a `Player` to release any exclusive resources and minimize its use of non-exclusive resources. Although buffering and memory management requirements for `Players` are not specified, most `Players` allocate buffers that are large by the standards of Java objects. A well-implemented `Player` releases as much internal memory as possible when `deallocate` is called.

The `deallocate` method can only be called on a *Stopped* `Player`. To avoid `ClockStartedErrors`, you should call `stop` before you call `deallocate`. Calling `deallocate` on a `Player` in the *Prefetching* or *Prefetched* state returns it to the *Realized* state. If `deallocate` is called while the `Player` is realizing, the `Player` posts a `DeallocateEvent` and returns to the *Unrealized* state. (Once a `Player` has been realized, it can never return to the *Unrealized* state.)

You generally call `deallocate` when the `Player` is not being used. For example, an applet should call `deallocate` as part of its `stop` method. By calling `deallocate`, the program can maintain references to the `Player`, while freeing other resources for use by the system as a whole. (JMF does not prevent a *Realized* `Player` that has formerly been *Prefetched* or *Started* from maintaining information that would allow it to be started up more quickly in the future.)

When you are finished with a `Player` (or any other `Controller`) and are not going to use it anymore, you should call `close`. The `close` method indicates that the `Controller` will no longer be used and can shut itself down. Calling `close` releases all of the resources that the `Controller` was using and causes it to cease all activity. When a `Controller` is closed, it posts a `ControllerClosedEvent`. A closed `Controller` cannot be reopened and invoking methods on a closed `Controller` might generate errors.

Getting the Time-Base Time

You can get a `Player` object's current time-base time by getting the `Player` object's `TimeBase` and calling `getTime`:

```
myCurrentTBTime = player1.getTimeBase().getTime();
```

When a `Player` is running, you can get the time-base time that corresponds to a particular media time by calling `mapToTimeBase`.

Getting the Duration of the Media Stream

Since programs often need to know how long a particular media stream will run, all `Controllers` implement the `Duration` interface. This interface defines a single method, `getDuration`. The duration represents the length of time that a media object would run, if played at the default rate of 1.0. A media stream's duration is only accessible through a `Player`.

If the duration can't be determined when `getDuration` is called, `DURATION_UNKNOWN` is returned. This can happen if the `Player` has not yet reached a state where the duration of the media source is available. At a later time, the duration might be available and a call to `getDuration` would return the duration value. If the media source does not have a defined duration, as in the case of a live broadcast, `getDuration` returns `DURATION_UNBOUNDED`.

Responding to Media Events

`ControllerListener` is an asynchronous interface for handling events generated by `Controller` objects. Using the `ControllerListener` interface enables you to manage the timing of potentially time-consuming `Player` operations such as prefetching.

Implementing the ControllerListener Interface

To implement the `ControllerListener` interface, you need to:

1. Implement the `ControllerListener` interface in a class.
2. Register that class as a listener by calling `addControllerListener` on the `Controller` that you want to receive events from.

When a `Controller` posts an event, it calls `controllerUpdate` on each registered listener.

Typically, `controllerUpdate` is implemented as a series of if-else statements.

Example 3-3:Implementing `controllerUpdate`.

```
if (event instanceof EventType){
...
} else if (event instanceof OtherEventType){
...
}
```

This filters out the events that you are not interested in. If you have registered as a listener with multiple `Controllers`, you also need to determine which `Controller` posted the event. `ControllerEvents` come "stamped" with a reference to their source that you can access by calling `getSource`.

When you receive events from a `Controller`, you might need to do some additional processing to ensure that the `Controller` is in the proper state before calling a control method. For example, before calling any of the methods that are restricted to *Stopped* `Players`, you should check the `Player` object's target state by calling `getTargetState`. If `start` has been called, the `Player` is considered to be in the *Started* state, though it might be posting transition events as it prepares the `Player` to present media.

Some types of `ControllerEvents` contain additional state information. For example, the `StartEvent` and `StopEvent` classes each define a method that allows you to retrieve the media time at which the event occurred.

Using ControllerAdapter

`ControllerAdapter` is a convenience class that implements `ControllerListener` and can be easily extended to respond to particular `Events`. To implement the `ControllerListener` interface using `ControllerAdapter`, you need to:

1. Subclass `ControllerAdapter` and override the event methods for the events that you're interested in.
2. Register your `ControllerAdapter` class as a listener for a particular `Controller` by calling `addControllerListener`.

When a Controller posts an event, it calls controllerUpdate on each registered listener. ControllerAdapter automatically dispatches the event to the appropriate event method, filtering out the events that you're not interested in.

For example, the following code extends a ControllerAdapter with a JDK 1.1 anonymous inner-class to create a self-contained Player that is automatically reset to the beginning of the media and deallocated when the Player reaches the end of the media.

Example 3-4: Using ControllerAdapter.

```
player.addControllerListener(new ControllerAdapter() {
    public void endOfMedia(EndOfMediaEvent e) {
        Controller controller = e.getSource();
        controller.stop();
        controller.setMediaTime(new Time(0));
        controller.deallocate();
    }
})
```

If you register a single ControllerAdapter as a listener for multiple Players, in your event method implementations you need to determine which Player generated the event. You can call getSource to determine where a ControllerEvent originated.

Synchronizing Multiple Media Streams

To synchronize the playback of multiple media streams, you can synchronize the Players by associating them with the same TimeBase. To do this, you use the getTimeBase and setTimeBase methods defined by the Clock interface. For example, you could synchronize player1 with player2 by setting player1 to use player2's time base:

```
player1.setTimeBase(player2.getTimeBase());
```

When you synchronize Players by associating them with the same TimeBase, you must still manage the control of each Player individually. Because managing synchronized Players in this way can be complicated, JMF provides a mechanism that allows a Player to assume control over any other Controller. The Player manages the states of these Controllers automatically, allowing you to interact with the entire group through a

single point of control. For more information, see See "Using a Player to Synchronize Controllers".

Using a Player to Synchronize Controllers

Synchronizing `Players` directly using `syncStart` requires that you carefully manage the states of all of the synchronized `Players`. You must control each one individually, listening for events and calling control methods on them as appropriate. Even with only a few `Players`, this quickly becomes a difficult task. Through the `Player` interface, JMF provides a simpler solution: a `Player` can be used to manage the operation of any `Controller`.

When you interact with a managing `Player`, your instructions are automatically passed along to the managed `Controllers` as appropriate. The managing `Player` takes care of the state management and synchronization for all of the other `Controllers`.

This mechanism is implemented through the `addController` and `removeController` methods. When you call `addController` on a `Player`, the `Controller` you specify is added to the list of `Controllers` managed by the `Player`. Conversely, when you call `removeController`, the specified `Controller` is removed from the list of managed `Controllers`.

Typically, when you need to synchronize `Players` or other `Controllers`, you should use this `addController` mechanism. It is simpler, faster, and less error-prone than attempting to manage synchronized `Players` individually.

When a `Player` assumes control of a `Controller`:

- The `Controller` assumes the `Player` object's time base.
- The `Player` object's duration becomes the longer of the `Controller` object's duration and its own. If multiple `Controllers` are placed under a `Player` object's control, the `Player` object's duration is set to longest duration.
- The `Player` object's start latency becomes the longer of the `Controller` object's start latency and its own. If multiple `Controllers` are placed under a `Player` object's control, the `Player` object's start latency is set to the longest latency.

A managing `Player` only posts completion events for asynchronous methods after each of its managed `Controllers` have posted the event. The

managing Player reposts other events generated by the Controllers as appropriate.

Adding a Controller

You use the addController method to add a Controller to the list of Controllers managed by a particular Player. To be added, a Controller must be in the *Realized* state; otherwise, a NotRealizedError is thrown. Two Players cannot be placed under control of each other. For example, if player1 is placed under the control of player2, player2 cannot be placed under the control of player1 without first removing player1 from player2's control.

Once a Controller has been added to a Player, do not call methods directly on the managed Controller. To control a managed Controller, you interact with the managing Player.

To have player2 assume control of player1, call:

```
player2.addController(player1);
```

Controlling Managed Controllers

To control the operation of a group of Controllers managed by a particular Player, you interact directly with the managing Player.

For example, to prepare all of the managed Controllers to start, call prefetch on the managing Player. Similarly, when you want to start them, call start on the managing Player. The managing Player makes sure that all of the Controllers are *Prefetched*, determines the maximum start latency among the Controllers, and calls syncStart to start them, specifying a time that takes the maximum start latency into account.

When you call a Controller method on the managing Player, the Player propagates the method call to the managed Controllers as appropriate. Before calling a Controller method on a managed Controller, the Player ensures that the Controller is in the proper state. The following table describes what happens to the managed Controllers when you call control methods on the managing Player.

Function	Stopped Player	Started Player
setMediaTime	Invokes setMediaTime on all managed Controllers.	Stops all managed Controllers, invokes setMediaTime, and restarts Controllers.
setRate	Invokes setRate on all managed Controllers. Returns the actual rate that was supported by all Controllers and set.	Stops all managed Controllers, invokes setRate, and restarts Controllers. Returns the actual rate that was supported by all Controllers and set.
start	Ensures all managed Controllers are *Prefetched* and invokes syncStart on each of them, taking into account their start latencies.	Depends on the Player implementation. Player might immediately post a StartEvent.
realize	The managing Player immediately posts a RealizeCompleteEvent. To be added, a Controller must already be realized.	The managing Player immediately posts a RealizeCompleteEvent. To be added, a Controller must already be realized.
prefetch	Invokes prefetch on all managed Controllers.	The managing Player immediately posts a PrefetchCompleteEvent, indicating that all managed Controllers are *Prefetched*.
stop	No effect.	Invokes stop on all managed Controllers.
deallocate	Invokes deallocate on all managed Controllers.	It is illegal to call deallocate on a *Started* Player.
setStopTime	Invokes setStopTime on all managed Controllers. (Player must be *Realized*.)	Invokes setStopTime on all managed Controllers. (Can only be set once on a *Started* Player.)
syncStart	Invokes syncStart on all managed Controllers.	It is illegal to call syncStart on a *Started* Player.
close	Invokes close on all managed Controllers.	It is illegal to call close on a *Started* Player.

Table 3-1: Calling control methods on a managing player.

Removing a Controller

You use the removeController method to remove a Controller from the list of controllers managed by a particular Player.

To have `player2` release control of `player1`, call:

```
player2.removeController(player1);
```

Synchronizing Players Directly

In a few situations, you might want to manage the synchronization of multiple `Player` objects yourself so that you can control the rates or media times independently. If you do this, you must:

1. Register as a listener for each synchronized `Player`.
2. Determine which `Player` object's time base is going to be used to drive the other `Player` objects and set the time base for the synchronized `Player` objects. Not all `Player objects` can assume a new time base. For example, if one of the `Player` objects you want to synchronize has a push data-source, that `Player` object's time base must be used to drive the other `Player objects`.
3. Set the rate for all of the `Players`. If a `Player` cannot support the rate you specify, it returns the rate that was used. (There is no mechanism for querying the rates that a `Player` supports.)
4. Synchronize the states of all of the `Player` objects. (For example, stop all of the players.)
5. Synchronize the operation of the `Player` objects:
    - Set the media time for each `Player`.
    - Prefetch each `Player`.
    - Determine the maximum start latency among the synchronized `Player` objects.
    - Start the `Player` objects by calling `syncStart` with a time that takes into account the maximum latency.

You must listen for transition events for all of the `Player objects` and keep track of which ones have posted events. For example, when you prefetch the `Player` objects, you need to keep track of which ones have posted `PrefetchComplete` events so that you can be sure all of them are *Prefetched* before calling `syncStart`. Similarly, when you request that the synchronized `Player` objects stop at a particular time, you need to listen

for the stop event posted by each `Player` to determine when all of them have actually stopped.

In some situations, you need to be careful about responding to events posted by the synchronized `Player` objects. To be sure of the state of all of the `Player` objects, you might need to wait at certain stages for all of them to reach the same state before continuing.

For example, assume that you are using one `Player` to drive a group of synchronized `Player` objects. A user interacting with that `Player` sets the media time to 10, starts the `Player`, and then changes the media time to 20. You then:

1. Pass along the first `setMediaTime` call to all of the synchronized `Player` objects.
2. Call `prefetch` on each `Player` to prepare them to start.
3. Call `stop` on each `Player` when the second set media time request is received.
4. Call `setMediaTime` on each `Player` with the new time.
5. Restart the prefetching operation.
6. When all of the `Player` objects have been prefetched, start them by calling `syncStart`, taking into account their start latencies.

In this case, just listening for `PrefetchComplete` events from all of the `Player` objects before calling `syncStart` isn't sufficient. You can't tell whether those events were posted in response to the first or second prefetch operation. To avoid this problem, you can block when you call `stop` and wait for all of the `Player` objects to post stop events before continuing. This guarantees that the next `PrefetchComplete` events you receive are the ones that you are really interested in.

Example: Playing an MPEG Movie in an Applet

The sample program `PlayerApplet` demonstrates how to create a `Player` and present an MPEG movie from within a Java applet. This is a general example that could easily be adapted to present other types of media streams.

The Player object's visual presentation and its controls are displayed within the applet's presentation space in the browser window. If you create a Player in a Java application, you are responsible for creating the window to display the Player object's components.

Note: While PlayerApplet illustrates the basic usage of a Player, it does not perform the error handling necessary in a real applet or application. For a more complete sample suitable for use as a template, see "JMF Applet" on page 173.

Overview of PlayerApplet

The APPLET tag is used to invoke PlayerApplet in an HTML file. The WIDTH and HEIGHT fields of the HTML APPLET tag determine the dimensions of the applet's presentation space in the browser window. The PARAM tag identifies the media file to be played.

Example 3-5: Invoking PlayerApplet.

```
<APPLET CODE=ExampleMedia.PlayerApplet
WIDTH=320 HEIGHT=300>
<PARAM NAME=FILE VALUE="sample2.mpg">
</APPLET>
```

When a user opens a web page containing PlayerApplet, the applet loads automatically and runs in the specified presentation space, which contains the Player object's visual component and default controls. The Player starts and plays the MPEG movie once. The user can use the default Player controls to stop, restart, or replay the movie. If the page containing the applet is closed while the Player is playing the movie, the Player automatically stops and frees the resources it was using.

To accomplish this, PlayerApplet extends Applet and implements the ControllerListener interface. PlayerApplet defines five methods:

- init—creates a Player for the file that was passed in through the PARAM tag and registers PlayerApplet as a controller listener so that it can observe media events posted by the Player. (This causes the PlayerApplet controllerUpdate method to be called whenever the Player posts an event.)
- start—starts the Player when PlayerApplet is started.
- stop—stops and deallocates the Player when PlayerApplet is stopped.
- destroy—closes the Player when PlayerApplet is removed.

- `controllerUpdate`—responds to `Player` events to display the `Player` object's components.

Example 3-6: `PlayerApplet`.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import javax.media.*;

public class PlayerApplet extends Applet implements ControllerListener {
   Player player = null;
   public void init() {
      setLayout(new BorderLayout());
      String mediaFile = getParameter("FILE");
      try {
         URL mediaURL = new URL(getDocumentBase(), mediaFile);
         player = Manager.createPlayer(mediaURL);
         player.addControllerListener(this);
      }
      catch (Exception e) {
         System.err.println("Got exception "+e);
      }
   }
   public void start() {
      player.start();
   }
   public void stop() {
      player.stop();
      player.deallocate();
   }
   public void destroy() {
      player.close();
   }
   public synchronized void controllerUpdate(ControllerEvent event) {
      if (event instanceof RealizeCompleteEvent) {
         Component comp;
         if ((comp = player.getVisualComponent()) != null)
            add ("Center", comp);
         if ((comp = player.getControlPanelComponent()) != null)
            add ("South", comp);
         validate();
      }
   }
}
```

Initializing the Applet

When a Java applet starts, its `init` method is invoked automatically. You override `init` to prepare your applet to be started. `PlayerApplet` performs four tasks in `init`:

1. Retrieves the applet's FILE parameter.
2. Uses the FILE parameter to locate the media file and build a `URL` object that describes that media file.
3. Creates a `Player` for the media file by calling `Manager.createPlayer`.
4. Registers the applet as a controller listener with the new `Player` by calling `addControllerListener`. Registering as a listener causes the `PlayerApplet` `controllerUpdate` method to be called automatically whenever the `Player` posts a media event. The `Player` posts media events whenever its state changes. This mechanism allows you to control the `Player` object's transitions between states and ensure that the `Player` is in a state in which it can process your requests. (For more information, see "Player States" on page 26.)

Example 3-7: Initializing `PlayerApplet`.

```
public void init() {
    setLayout(new BorderLayout());
    // 1. Get the FILE parameter.
    String mediaFile = getParameter("FILE");
    try {
        // 2. Create a URL from the FILE parameter. The URL
        // class is defined in java.net.
        URL mediaURL = new URL(getDocumentBase(), mediaFile);
        // 3. Create a player with the URL object.
        player = Manager.createPlayer(mediaURL);
        // 4. Add PlayerApplet as a listener on the new player.
        player.addControllerListener(this);
    }
    catch (Exception e) {
        System.err.println("Got exception "+e);
    }
}
```

Controlling the Player

The Applet class defines start and stop methods that are called automatically when the page containing the applet is opened and closed. You override these methods to define what happens each time your applet starts and stops.

PlayerApplet implements start to start the Player whenever the applet is started.

Example 3-8: Starting the Player in PlayerApplet.

```
public void start() {
    player.start();
}
```

Similarly, PlayerApplet overrides stop to stop and deallocate the Player:

Example 3-9: Stopping the Player in PlayerApplet.

```
public void stop() {
    player.stop();
    player.deallocate();
}
```

Deallocating the Player releases any resources that would prevent another Player from being started. For example, if the Player uses a hardware device to present its media, deallocate frees that device so that other Players can use it.

When an applet exits, destroy is called to dispose of any resources created by the applet. PlayerApplet overrides destroy to close the Player. Closing a Player releases all of the resources that it's using and shuts it down permanently.

Example 3-10: Destroying the Player in PlayerApplet.

```
public void destroy() {
    player.close();
}
```

Responding to Media Events

PlayerApplet registers itself as a ControllerListener in its init method so that it receives media events from the Player. To respond to these events, PlayerApplet implements the controllerUpdate method, which is called automatically when the Player posts an event.

PlayerApplet responds to one type of event, RealizeCompleteEvent. When the Player posts a RealizeCompleteEvent, PlayerApplet displays the Player object's components.

Example 3-11: Responding to media events in PlayerApplet.

```
public synchronized void controllerUpdate(ControllerEvent event)
{
    if (event instanceof RealizeCompleteEvent) {
        Component comp;
        if ((comp = player.getVisualComponent()) != null)
            add ("Center", comp);
        if ((comp = player.getControlPanelComponent()) != null)
            add ("South", comp);
        validate();
    }
}
```

A Player object's user-interface components cannot be displayed until the Player is *Realized*; an *Unrealized* Player doesn't know enough about its media stream to provide access to its user-interface components. PlayerApplet waits for the Player to post a RealizeCompleteEvent and then displays the Player object's visual component and default control panel by adding them to the applet container. Calling validate triggers the layout manager to update the display to include the new components.

Presenting Media with the MediaPlayer Bean

Using the MediaPlayer Java Bean (javax.media.bean.playerbean.MediaPlayer) is the simplest way to present media streams in your applets and applications. MediaPlayer encapsulates a full-featured JMF Player in a Java Bean. You can either use the MediaPlayer bean's default controls or customize its control Components.

One key advantage to using the MediaPlayer bean is that it automatically constructs a new Player when a different media stream is selected for

7

Working with Real-Time Media Streams

To send or receive a live media broadcast or conduct a video conference over the Internet or Intranet, you need to be able to receive and transmit media streams in real-time. This chapter introduces streaming media concepts and describes the Real-time Transport Protocol JMF uses for receiving and transmitting media streams across the network.

Streaming Media

When media content is streamed to a client in real-time, the client can begin to play the stream without having to wait for the complete stream to download. In fact, the stream might not even have a predefined duration—downloading the entire stream before playing it would be impossible. The term *streaming media* is often used to refer to both this technique of delivering content over the network in real-time and the real-time media content that's delivered.

Streaming media is everywhere you look on the web—live radio and television broadcasts and webcast concerts and events are being offered by a rapidly growing number of web portals, and it's now possible to conduct audio and video conferences over the Internet. By enabling the delivery of dynamic, interactive media content across the network, streaming media is changing the way people communicate and access information.

Protocols for Streaming Media

Transmitting media data across the net in real-time requires high network throughput. It's easier to compensate for lost data than to compensate for

large delays in receiving the data. This is very different from accessing static data such as a file, where the most important thing is that all of the data arrive at its destination. Consequently, the protocols used for static data don't work well for streaming media.

The HTTP and FTP protocols are based on the Transmission Control Protocol (TCP). TCP is a transport-layer protocol[1] designed for reliable data communications on low-bandwidth, high-error-rate networks. When a packet is lost or corrupted, it's retransmitted. The overhead of guaranteeing reliable data transfer slows the overall transmission rate.

For this reason, underlying protocols other than TCP are typically used for streaming media. One that's commonly used is the User Datagram Protocol (UDP). UDP is an unreliable protocol; it does not guarantee that each packet will reach its destination. There's also no guarantee that the packets will arrive in the order that they were sent. The receiver has to be able to compensate for lost data, duplicate packets, and packets that arrive out of order.

Like TCP, UDP is a general transport-layer protocol—a lower-level networking protocol on top of which more application-specific protocols are built. The Internet standard for transporting real-time data such as audio and video is the Real-Time Transport Protocol (RTP).

RTP is defined in IETF RFC 1889, a product of the AVT working group of the Internet Engineering Task Force (IETF).

Real-Time Transport Protocol

RTP provides end-to-end network delivery services for the transmission of real-time data. RTP is network and transport-protocol independent, though it is often used over UDP.

1. In the seven layer ISO/OSI data communications model, the transport layer is level four. For more information about the ISO/OSI model, see *Understanding OSI*. Larmouth, John. International Thompson Computer Press, 1996. ISBN 1850321760.

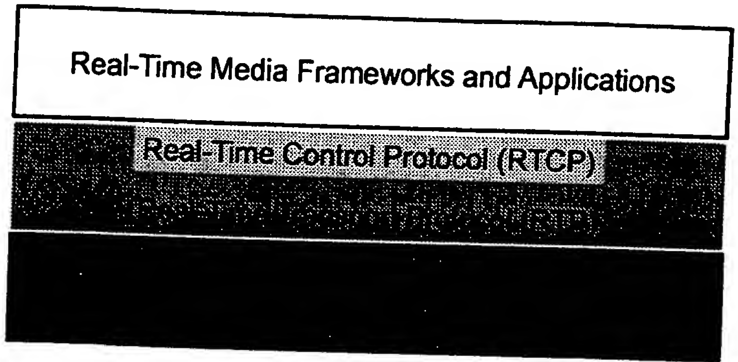


Figure 7-1: RTP architecture.

RTP can be used over both unicast and multicast network services. Over a *unicast* network service, separate copies of the data are sent from the source to each destination. Over a *multicast* network service, the data is sent from the source only once and the network is responsible for transmitting the data to multiple locations. Multicasting is more efficient for many multimedia applications, such as video conferences. The standard Internet Protocol (IP) supports multicasting.

RTP Services

RTP enables you to identify the type of data being transmitted, determine what order the packets of data should be presented in, and synchronize media streams from different sources.

RTP data packets are not guaranteed to arrive in the order that they were sent—in fact, they're not guaranteed to arrive at all. It's up to the receiver to reconstruct the sender's packet sequence and detect lost packets using the information provided in the packet header.

While RTP does not provide any mechanism to ensure timely delivery or provide other quality of service guarantees, it is augmented by a control protocol (RTCP) that enables you to monitor the quality of the data distribution. RTCP also provides control and identification mechanisms for RTP transmissions.

If quality of service is essential for a particular application, RTP can be used over a resource reservation protocol that provides connection-oriented services.

RTP Architecture

An RTP *session* is an association among a set of applications communicating with RTP. A session is identified by a network address and a pair of ports. One port is used for the media data and the other is used for control (RTCP) data.

A *participant* is a single machine, host, or user participating in the session. Participation in a session can consist of passive reception of data (receiver), active transmission of data (sender), or both.

Each media type is transmitted in a different session. For example, if both audio and video are used in a conference, one session is used to transmit the audio data and a separate session is used to transmit the video data. This enables participants to choose which media types they want to receive—for example, someone who has a low-bandwidth network connection might only want to receive the audio portion of a conference.

Data Packets

The media data for a session is transmitted as a series of packets. A series of data packets that originate from a particular source is referred to as an *RTP stream*. Each RTP data packet in a stream contains two parts, a structured header and the actual data (the packet's *payload*).

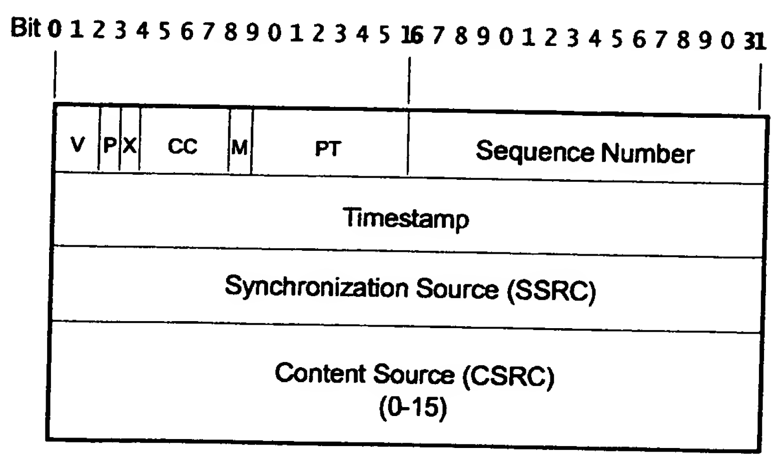


Figure 7-2: RTP data-packet header format.

The header of an RTP data packet contains:

- **The RTP version number (V)**: 2 bits. The version defined by the current specification is 2.

- **Padding** (P): 1 bit. If the padding bit is set, there are one or more bytes at the end of the packet that are not part of the payload. The very last byte in the packet indicates the number of bytes of padding. The padding is used by some encryption algorithms.
- **Extension** (X): 1 bit. If the extension bit is set, the fixed header is followed by one header extension. This extension mechanism enables implementations to add information to the RTP Header.
- **CSRC Count** (CC): 4 bits. The number of CSRC identifiers that follow the fixed header. If the CSRC count is zero, the synchronization source is the source of the payload.
- **Marker (M)**: 1 bit. A marker bit defined by the particular media profile.
- **Payload Type** (PT): 7 bits. An index into a media profile table that describes the payload format. The payload mappings for audio and video are specified in RFC 1890.
- **Sequence Number**: 16 bits. A unique packet number that identifies this packet's position in the sequence of packets. The packet number is incremented by one for each packet sent.
- **Timestamp**: 32 bits. Reflects the sampling instant of the first byte in the payload. Several consecutive packets can have the same timestamp if they are logically generated at the same time—for example, if they are all part of the same video frame.
- **SSRC**: 32 bits. Identifies the synchronization source. If the CSRC count is zero, the payload source is the synchronization source. If the CSRC count is nonzero, the SSRC identifies the mixer.
- **CSRC**: 32 bits each. Identifies the contributing sources for the payload. The number of contributing sources is indicated by the CSRC count field; there can be up to 16 contributing sources. If there are multiple contributing sources, the payload is the mixed data from those sources.

Control Packets

In addition to the media data for a session, control data (RTCP) packets are sent periodically to all of the participants in the session. RTCP packets can contain information about the quality of service for the session participants, information about the source of the media being transmitted on the data port, and statistics pertaining to the data that has been transmitted so far.

There are several types of RTCP packets:

- Sender Report
- Receiver Report
- Source Description
- Bye
- Application-specific

RTCP packets are "stackable" and are sent as a compound packet that contains at least two packets, a report packet and a source description packet.

All participants in a session send RTCP packets. A participant that has recently sent data packets issues a *sender report*. The sender report (SR) contains the total number of packets and bytes sent as well as information that can be used to synchronize media streams from different sessions.

Session participants periodically issue *receiver reports* for all of the sources from which they are receiving data packets. A receiver report (RR) contains information about the number of packets lost, the highest sequence number received, and a timestamp that can be used to estimate the round-trip delay between a sender and the receiver.

The first packet in a compound RTCP packet has to be a report packet, even if no data has been sent or received—in which case, an empty receiver report is sent.

All compound RTCP packets must include a source description (SDES) element that contains the canonical name (CNAME) that identifies the source. Additional information might be included in the source description, such as the source's name, email address, phone number, geographic location, application name, or a message describing the current state of the source.

When a source is no longer active, it sends an RTCP BYE packet. The BYE notice can include the reason that the source is leaving the session.

RTCP APP packets provide a mechanism for applications to define and send custom information via the RTP control port.

RTP Applications

RTP applications are often divided into those that need to be able to receive data from the network (RTP Clients) and those that need to be able

to transmit data across the network (RTP Servers). Some applications do both—for example, conferencing applications capture and transmit data at the same time that they're receiving data from the network.

Receiving Media Streams From the Network

Being able to receive RTP streams is necessary for several types of applications. For example:

- Conferencing applications need to be able to receive a media stream from an RTP session and render it on the console.
- A telephone answering machine application needs to be able to receive a media stream from an RTP session and store it in a file.
- An application that records a conversation or conference must be able to receive a media stream from an RTP session and both render it on the console and store it in a file.

Transmitting Media Streams Across the Network

RTP server applications transmit captured or stored media streams across the network.

For example, in a conferencing application, a media stream might be captured from a video camera and sent out on one or more RTP sessions. The media streams might be encoded in multiple media formats and sent out on several RTP sessions for conferencing with heterogeneous receivers. Multiparty conferencing could be implemented without IP multicast by using multiple unicast RTP sessions.

References

The RTP specification is a product of the Audio Video Transport (AVT) working group of the Internet Engineering Task Force (IETF). For additional information about the IETF, see `http://www.ietf.org`. The AVT working group charter and proceedings are available at `http://www.ietf.org/html.charters/avt-charter.html`.

IETF RFC 1889, RTP: A Transport Protocol for Real Time Applications
Current revision: `http://www.ietf.org.internet-drafts/draft-ietf-avt-rtp-new-04.txt`

IETF RFC 1890: RTP Profile for Audio and Video Conferences with Minimal Control

Current revision: `http://www.ietf.org.internet-drafts/draft-ietf-avt-profile-new-06.txt`

Note: These RFCs are undergoing revisions in preparation for advancement from Proposed Standard to Draft Standard and the URLs listed here are for the Internet Drafts of the revisions available at the time of publication.

In addition to these RFCs, separate payload specification documents define how particular payloads are to be carried in RTP. For a list of all of the RTP-related specifications, see the AVT working group charter at: `http://www.ietf.org/html.charters/avt-charter.html`.

8

Understanding the JMF RTP API

JMF enables the playback and transmission of RTP streams through the APIs defined in the `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp` packages. JMF can be extended to support additional RTP-specific formats and dynamic payloads through the standard JMF plug-in mechanism.

Note: JMF-compliant implementations are not required to support the RTP APIs in `javax.media.rtp`, `javax.media.rtp.event`, and `javax.media.rtp.rtcp`. The reference implementations of JMF provided by Sun Microsystems, Inc. and IBM Corporation fully support these APIs.

You can play incoming RTP streams locally, save them to a file, or both.

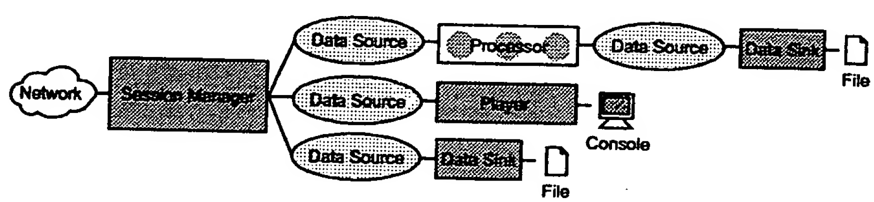


Figure 8-1: RTP reception.

For example, the RTP APIs could be used to implement a telephony application that answers calls and records messages like an answering machine.

Similarly, you can use the RTP APIs to transmit captured or stored media streams across the network. Outgoing RTP streams can originate from a file or a capture device. The outgoing streams can also be played locally, saved to a file, or both.

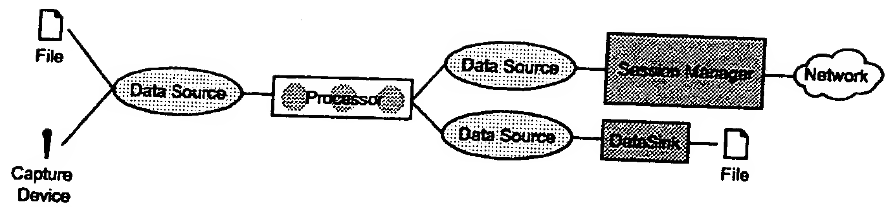


Figure 8-2: RTP transmission.

For example, you could implement a video conferencing application that captures live audio and video and transmits it across the network using a separate RTP session for each media type.

Similarly, you might record a conference for later broadcast or use a prerecorded audio stream as "hold music" in a conferencing application.

RTP Architecture

The JMF RTP APIs are designed to work seamlessly with the capture, presentation, and processing capabilities of JMF. Players and processors are used to present and manipulate RTP media streams just like any other media content. You can transmit media streams that have been captured from a local capture device using a capture `DataSource` or that have been stored to a file using a `DataSink`. Similarly, JMF can be extended to support additional RTP formats and payloads through the standard plug-in mechanism.

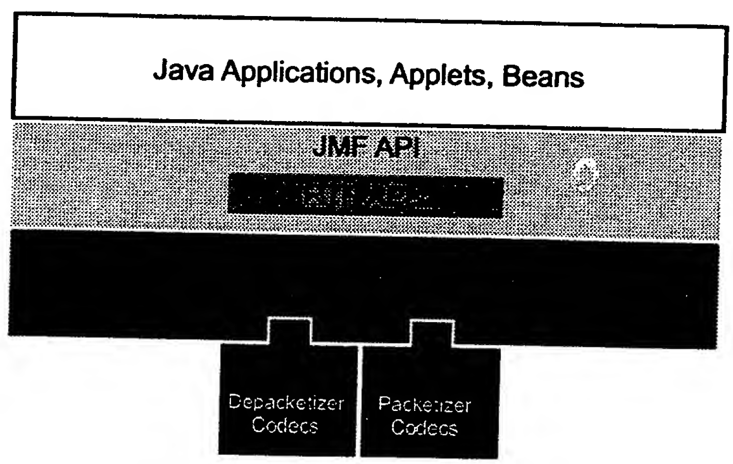


Figure 8-3: High-level JMF RTP architecture.

Session Manager

In JMF, a `SessionManager` is used to coordinate an RTP session. The session manager keeps track of the session participants and the streams that are being transmitted.

The session manager maintains the state of the session as viewed from the local participant. In effect, a session manager is a local representation of a distributed entity, the RTP session. The session manager also handles the *RTCP* control channel, and supports RTCP for both senders and receivers.

The `SessionManager` interface defines methods that enable an application to initialize and start participating in a session, remove individual streams created by the application, and close the entire session.

Session Statistics

The session manager maintains statistics on all of the RTP and RTCP packets sent and received in the session. Statistics are tracked for the entire session on a per-stream basis. The session manager provides access to global reception and transmission statistics:

- **`GlobalReceptionStats`**: Maintains global reception statistics for the session.
- **`GlobalTransmissionStats`**: Maintains cumulative transmission statistics for all local senders.

Statistics for a particular recipient or outgoing stream are available from the stream:

- **`ReceptionStats`**: Maintains source reception statistics for an individual participant.
- **`TransmissionStats`**: Maintains transmission statistics for an individual send stream.

Session Participants

The session manager keeps track of all of the participants in a session. Each participant is represented by an instance of a class that implements the `Participant` interface. `SessionManagers` create a `Participant` whenever an RTCP packet arrives that contains a source description (SDES) with a canonical name (`CNAME`) that has not been seen before in the session (or has timed-out since its last use). Participants can be passive (sending

control packets only) or active (also sending one or more RTP data streams).

There is exactly one *local participant* that represents the local client/server participant. A local participant indicates that it will begin sending RTCP control messages or data and maintain state on incoming data and control messages by starting a session.

A participant can own more than one stream, each of which is identified by the synchronization source identifier (SSRC) used by the source of the stream.

Session Streams

The `SessionManager` maintains an `RTPStream` object for each stream of RTP data packets in the session. There are two types of RTP streams:

- `ReceiveStream` represents a stream that's being received from a remote participant.
- `SendStream` represents a stream of data coming from the `Processor` or input `DataSource` that is being sent over the network.

A `ReceiveStream` is constructed automatically whenever the session manager detects a new source of RTP data. To create a new `SendStream`, you call the `SessionManager` `createSendStream` method.

RTP Events

Several RTP-specific events are defined in `javax.media.rtp.event`. These events are used to report on the state of the RTP session and streams.

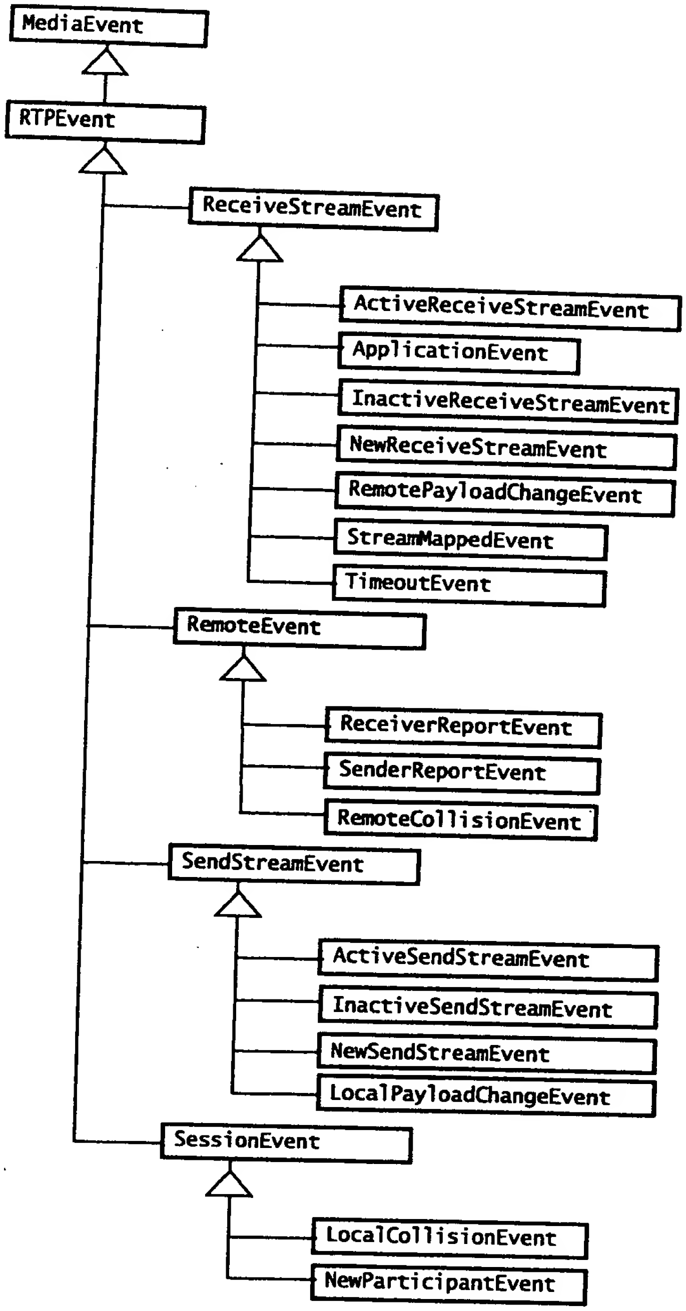


Figure 8-4: RTP events.

To receive notification of RTP events, you implement the appropriate RTP listener and register it with the session manager:

- `SessionListener`: Receives notification of changes in the state of the session.

- `SendStreamListener`: Receives notification of changes in the state of an RTP stream that's being transmitted.
- `ReceiveStreamListener`: Receives notification of changes in the state of an RTP stream that's being received.
- `RemoteListener`: Receives notification of events or RTP control messages received from a remote participant.

Session Listener

You can implement `SessionListener` to receive notification about events that pertain to the RTP session as a whole, such as the addition of new participants.

There are two types of session-wide events:

- `NewParticipantEvent`: Indicates that a new participant has joined the session.
- `LocalCollisionEvent`: Indicates that the participant's synchronization source is already in use.

Send Stream Listener

You can implement `SendStreamListener` to receive notification whenever:

- New send streams are created by the local participant.
- The transfer of data from the `DataSource` used to create the send stream has started or stopped.
- The send stream's format or payload changes.

There are five types of events associated with a `SendStream`:

- `NewSendStreamEvent`: Indicates that a new send stream has been created by the local participant.
- `ActiveSendStreamEvent`: Indicates that the transfer of data from the `DataSource` used to create the send stream has started.
- `InactiveSendStreamEvent`: Indicates that the transfer of data from the DataSource used to create the send stream has stopped.
- `LocalPayloadChangeEvent`: Indicates that the stream's format or payload has changed.

- `StreamClosedEvent`: Indicates that the stream has been closed.

Receive Stream Listener

You can implement `ReceiveStreamListener` to receive notification whenever:

- New receive streams are created.
- The transfer of data starts or stops.
- The data transfer times out.
- A previously orphaned `ReceiveStream` has been associated with a `Participant`.
- An RTCP APP packet is received.
- The receive stream's format or payload changes.

You can also use this interface to get a handle on the stream and access the RTP `DataSource` so that you can create a `MediaHandler`.

There are seven types of events associated with a `ReceiveStream`:

- `NewReceiveStreamEvent`: Indicates that the session manager has created a new receive stream for a newly-detected source.
- `ActiveReceiveStreamEvent`: Indicates that the transfer of data has started.
- `InactiveReceiveStreamEvent`: Indicates that the transfer of data has stopped.
- `TimeoutEvent`: Indicates that the data transfer has timed out.
- `RemotePayloadChangeEvent`: Indicates that the format or payload of the receive stream has changed.
- `StreamMappedEvent`: Indicates that a previously orphaned receive stream has been associated with a participant.
- `ApplicationEvent`: Indicates that an RTCP APP packet has been received.

Remote Listener

You can implement `RemoteListener` to receive notification of events or RTP control messages received from a remote participants. You might want to implement `RemoteListener` in an application used to monitor the

session—it enables you to receive RTCP reports and monitor the quality of the session reception without having to receive data or information on each stream.

There are three types of events associated with a remote participant:

- **ReceiverReportEvent**: Indicates that an RTP receiver report has been received.
- **SenderReportEvent**: Indicates that an RTP sender report has been received.
- **RemoteCollisionEvent**: Indicates that two remote participants are using the same synchronization source ID (SSRC).

RTP Data

The streams within an RTP session are represented by RTPStream objects. There are two types of RTPStreams: ReceiveStream and SendStream. Each RTP stream has a buffer data source associated with it. For ReceiveStreams, this DataSource is always a PushBufferDataSource.

The session manager automatically constructs new receive streams as it detects additional streams arriving from remote participants. You construct new send streams by calling createSendStream on the session manager.

Data Handlers

The JMF RTP APIs are designed to be transport-protocol independent. A custom RTP data handler can be created to enable JMF to work over a specific transport protocol. The data handler is a DataSource that can be used as the media source for a Player.

The abstract class RTPPushDataSource defines the basic elements of a JMF RTP data handler. A data handler has both an input data stream (PushSourceStream) and an output data stream (OuputDataStream). A data handler can be used for either the data channel or the control channel of an RTP session. If it is used for the data channel, the data handler implements the DataChannel interface.

An RTPSocket is an RTPPushDataSource has both a data and control channel. Each channel has an input and output stream to stream data to and from the underlying network. An RTPSocket can export RTPControls to add dynamic payload information to the session manager.

Because a custom `RTPSocket` can be used to construct a `Player` through the `Manager`, JMF defines the name and location for custom `RTPSocket` implementations:

```
<protocol package-prefix>.media.protocol.rtpraw.DataSource
```

RTP Data Formats

All RTP-specific data uses an RTP-specific format encoding as defined in the `AudioFormat` and `VideoFormat` classes. For example, gsm RTP encapsulated packets have the encoding set to `AudioFormat.GSM_RTP`, while jpeg-encoded video formats have the encoding set to `VideoFormat.JPEG_RTP`.

`AudioFormat` defines four standard RTP-specific encoding strings:

```
public static final String ULAW_RTP = "JAUDIO_G711_ULAW/rtp";
public static final String DVI_RTP = "dvi/rtp";
public static final String G723_RTP = "g723/rtp";
public static final String GSM_RTP = "gsm/rtp";
```

`VideoFormat` defines three standard RTP-specific encoding strings:

```
public static final String JPEG_RTP = "jpeg/rtp";
public static final String H261_RTP = "h261/rtp";
public static final String H263_RTP = "h263/rtp";
```

RTP Controls

The RTP API defines one RTP-specific control, `RTPControl`. `RTPControl` is typically implemented by RTP-specific `DataSources`. It provides a mechanism to add a mapping between a dynamic payload and a `Format`. `RTPControl` also provides methods for accessing session statistics and getting the current payload `Format`.

`SessionManager` also extends the `Controls` interface, enabling a session manager to export additional `Controls` through the `getControl` and `getControls` methods. For example, the session manager can export a `BufferControl` to enable you to specify the buffer length and threshold.

Reception

The presentation of an incoming RTP stream is handled by a `Player`. To receive and present a single stream from an RTP session, you can use a

MediaLocator that describes the session to construct a Player. A media locator for an RTP session is of the form:

```
rtp://address:port[:ssrc]/content-type/[ttl]
```

The Player is constructed and connected to the first stream in the session.

If there are multiple streams in the session that you want to present, you need to use a session manager. You can receive notification from the session manager whenever a stream is added to the session and construct a Player for each new stream. Using a session manager also enables you to directly monitor and control the session.

Transmission

A session manager can also be used to initialize and control a session so that you can stream data across the network. The data to be streamed is acquired from a Processor.

For example, to create a send stream to transmit data from a live capture source, you would:

1. Create, initialize, and start a SessionManager for the session.
2. Construct a Processor using the appropriate capture DataSource.
3. Set the output format of the Processor to an RTP-specific format. An appropriate RTP packetizer codec must be available for the data format you want to transmit.
4. Retrieve the output DataSource from the Processor.
5. Call createSendStream on the session manager and pass in the DataSource.

You control the transmission through the SendStream start and stop methods.

When it is first started, the SessionManager behaves as a receiver (sends out RTCP receiver reports). As soon as a SendStream is created, it begins to send out RTCP sender reports and behaves as a sender host as long as one or more send streams exist. If all SendStreams are closed (not just stopped), the session manager reverts to being a passive receiver.

Extensibility

Like the other parts of JMF, the RTP capabilities can be enhanced and extended. The RTP APIs support a basic set of RTP formats and payloads. Advanced developers and technology providers can implement JMF plug-ins to support dynamic payloads and additional RTP formats.

Implementing Custom Packetizers and Depacketizers

To implement a custom packetizer or depacketizer, you implement the JMF `Codec` interface. (For general information about JMF plug-ins, see "Implementing JMF Plug-Ins" on page 85.)

9

Receiving and Presenting RTP Media Streams

JMF Players and Processors provide the presentation, capture, and data conversion mechanisms for RTP streams.

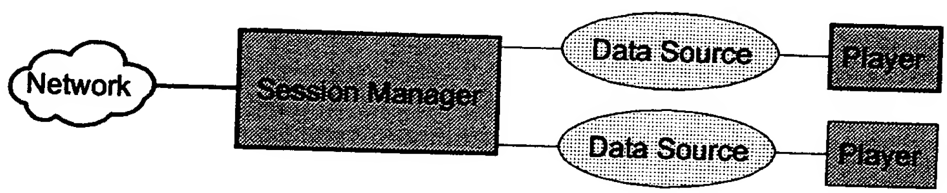


Figure 9-1: RTP reception data flow.

A separate player is used for each stream received by the session manager. You construct a Player for an RTP stream through the standard Manager createPlayer mechanism. You can either:

- Use a MediaLocator that has the parameters of the RTP session and construct a Player by calling Manager.createPlayer(MediaLocator)
- Construct a Player for a particular ReceiveStream by retrieving the DataSource from the stream and passing it to Manager.createPlayer(DataSource).

If you use a MediaLocator to construct a Player, you can only present the first RTP stream that's detected in the session. If you want to play back multiple RTP streams in a session, you need to use the SessionManager directly and construct a Player for each ReceiveStream.

Creating a Player for an RTP Session

When you use a `MediaLocator` to construct a `Player` for an RTP session, the `Manager` creates a `Player` for the first stream detected in the session. This `Player` posts a `RealizeCompleteEvent` once data has been detected in the session.

By listening for the `RealizeCompleteEvent`, you can determine whether or not any data has arrived and if the `Player` is capable of presenting any data. Once the `Player` posts this event, you can retrieve its visual and control components.

Note: Because a `Player` for an RTP media stream doesn't finish realizing until data is detected in the session, you shouldn't try to use `Manager.createRealizedPlayer` to construct a `Player` for an RTP media stream. No `Player` would be returned until data arrives and if no data is detected, attempting to create a *Realized* `Player` would block indefinitely.

A `Player` can export one RTP-specific control, `RTPControl`, which provides overall session statistics and can be used for registering dynamic payloads with the `SessionManager`.

Example 9-1: Creating a `Player` for an RTP session (1 of 2)

```
String url= "rtp://224.144.251.104:49150/audio/1";

MediaLocator mrl= new MediaLocator(url);

if (mrl == null) {
    System.err.println("Can't build MRL for RTP");
    return false;
}

// Create a player for this rtp session
try {
    player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
    System.err.println("Error:" + e);
    return false;
} catch (MalformedURLException e) {
    System.err.println("Error:" + e);
    return false;
} catch (IOException e) {
    System.err.println("Error:" + e);
    return false;
}

if (player != null) {
    if (this.player == null) {
```

Example 9-1: Creating a `Player` for an RTP session (2 of 2)

```
                this.player = player;
                player.addControllerListener(this);
                player.realize();
            }
        }
```

Listening for Format Changes

When a `Player` posts a `FormatChangeEvent`, it might indicate that a payload change has occurred. `Players` constructed with a `MediaLocator` automatically process payload changes. In most cases, this processing involves constructing a new `Player` to handle the new format. Applications that present RTP media streams need to listen for `FormatChangeEvents` so that they can respond if a new `Player` is created.

When a `FormatChangeEvent` is posted, check whether or not the `Player` object's control and visual components have changed. If they have, a new `Player` has been constructed and you need to remove references to the old `Player` object's components and get the new `Player` object's components.

Example 9-2: Listening for RTP format changes (1 of 2)

```
public synchronized void controllerUpdate(ControllerEvent ce) {
    if (ce instanceof FormatChangeEvent) {
        Dimension vSize = new Dimension(320,0);
        Component oldVisualComp = visualComp;

        if ((visualComp = player.getVisualComponent()) != null) {
            if (oldVisualComp != visualComp) {
                if (oldVisualComp != null) {
                    oldVisualComp.remove(zoomMenu);
                }

                framePanel.remove(oldVisualComp);

                vSize = visualComp.getPreferredSize();
                vSize.width = (int)(vSize.width * defaultScale);
                vSize.height = (int)(vSize.height * defaultScale);

                framePanel.add(visualComp);

                visualComp.setBounds(0,
                                     0,
                                     vSize.width,
                                     vSize.height);
                addPopupMenu(visualComp);
        }
```

Example 9-2: Listening for RTP format changes (2 of 2)

```
            }

            Component oldComp = controlComp;

            controlComp = player.getControlPanelComponent();

            if (controlComp != null)
            {
                if (oldComp != controlComp)
                {
                    framePanel.remove(oldComp);
                    framePanel.add(controlComp);

                    if (controlComp != null) {
                        int prefHeight = controlComp
                                       .getPreferredSize()
                                       .height;

                        controlComp.setBounds(0,
                                              vSize.height,
                                              vSize.width,
                                              prefHeight);
                    }
                }
            }
        }
    }
```

Creating an RTP Player for Each New Receive Stream

To play all of the `ReceiveStreams` in a session, you need to create a separate `Player` for each stream. When a new stream is created, the session manager posts a `NewReceiveStreamEvent`. Generally, you register as a `ReceiveStreamListener` and construct a `Player` for each new `ReceiveStream`. To construct the `Player`, you retrieve the `DataSource` from the `ReceiveStream` and pass it to `Manager.createPlayer`.

To create a `Player` for each new receive stream in a session:

1. Set up the RTP session:
    a. Create a `SessionManager`. For example, construct an instance of `com.sun.media.rtp.RTPSessionMgr`. (`RTPSessionMgr` is an implementation of `SessionManager` provided with the JMF reference implementation.)

b. Call `RTPSessionMgr addReceiveStreamListener` to register as a listener.

c. Initialize the RTP session by calling `RTPSessionMgr initSession`.

d. Start the RTP session by calling `RTPSessionMgr startSession`.

Example 9-3: Setting up an RTP session (1 of 2)

```
public SessionManager createManager(String address,
                                    int port,
                                    int ttl,
                                    boolean listener,
                                    boolean sendlistener)
{
    mgr = (SessionManager)new com.sun.media.rtp.RTPSessionMgr();

    if (mgr == null) return null;

    mgr.addFormat(new AudioFormat(AudioFormat.DVI_RTP,
                                  44100,
                                  4,
                                  1),
                  18);

    if (listener) mgr.addReceiveStreamListener(this);
    if (sendlistener) new RTPSendStreamWindow(mgr);

    // ask session mgr to generate the local participant's CNAME
    String cname = mgr.generateCNAME();
    String username = null;

    try {
        username = System.getProperty("user.name");
    } catch (SecurityException e){
        username = "jmf-user";
    }

    // create our local Session Address
    SessionAddress localaddr = new SessionAddress();

    try{
        InetAddress destaddr = InetAddress.getByName(address);

        SessionAddress sessaddr = new SessionAddress(destaddr,
                                                     port,
                                                     destaddr,
                                                     port + 1);
        SourceDescription[] userdesclist= new SourceDescription[]
        {
            new SourceDescription(SourceDescription
                                  .SOURCE_DESC_EMAIL,
                                  "jmf-user@sun.com",
```

Example 9-3: Setting up an RTP session (2 of 2)

```
                                           1,
                                           false),

                    new SourceDescription(SourceDescription
                                           .SOURCE_DESC_CNAME,
                                           cname,
                                           1,
                                           false),

                    new SourceDescription(SourceDescription
                                           .SOURCE_DESC_TOOL,
                                           "JMF RTP Player v2.0",
                                           1,
                                           false)
                };

                mgr.initSession(localaddr,
                                userdesclist,
                                0.05,
                                0.25);

                mgr.startSession(sessaddr,ttl,null);
            } catch (Exception e) {
                System.err.println(e.getMessage());
                return null;
            }

            return mgr;
        }
```

2. In your `ReceiveStreamListener` `update` method, watch for `NewReceiveStreamEvent`, which indicates that a new data stream has been detected.

3. When a `NewReceiveStreamEvent` is detected, retrieve the `ReceiveStream` from the `NewReceiveStreamEvent` by calling `getReceiveStream`.

4. Retrieve the RTP `DataSource` from the `ReceiveStream` by calling `getDataSource`. This is a `PushBufferDataSource` with an RTP-specific `Format`. For example, the encoding for a DVI audio player will be DVI_RTP.

5. Pass the `DataSource` to `Manager.createPlayer` to construct a `Player`. For the `Player` to be successfully constructed, the necessary plug-ins for decoding and depacketizing the RTP-formatted data must be available. (For more information, see "Creating Custom Packetizers and Depacketizers" on page 167).

Example 9-4: Listening for `NewReceiveStreamEvents`

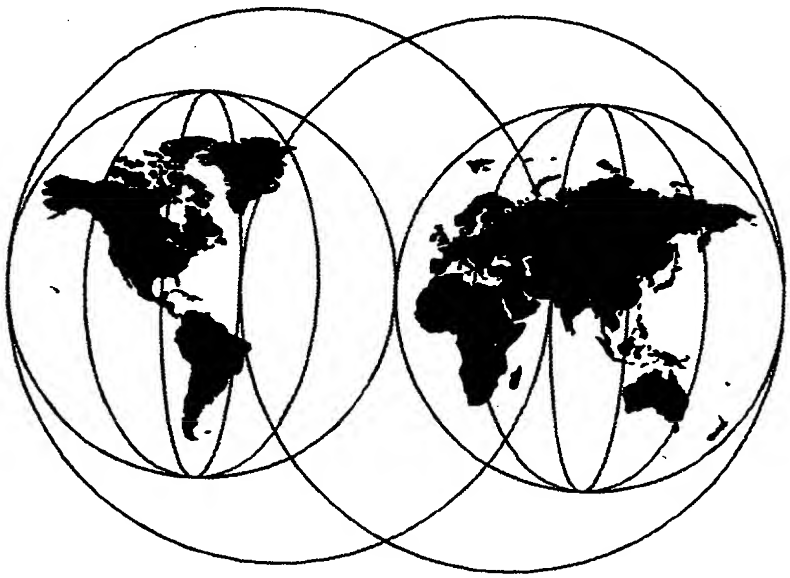
```
public void update( ReceiveStreamEvent event)
{
    Player newplayer = null;
    RTPPlayerWindow playerWindow = null;

    // find the sourceRTPSM for this event
    SessionManager source = (SessionManager)event.getSource();

    // create a new player if a new recvstream is detected
    if (event instanceof NewReceiveStreamEvent)
    {
        String cname = "Java Media Player";
        ReceiveStream stream = null;

        try
        {
            // get a handle over the ReceiveStream
            stream =((NewReceiveStreamEvent)event)
                    .getReceiveStream();

            Participant part = stream.getParticipant();

            if (part != null) cname = part.getCNAME();

            // get a handle over the ReceiveStream datasource
            DataSource dsource = stream.getDataSource();

            // create a player by passing datasource to the
            // Media Manager
            newplayer = Manager.createPlayer(dsource);
            System.out.println("created player " + newplayer);
        } catch (Exception e) {
            System.err.println("NewReceiveStreamEvent exception "
                               + e.getMessage());
            return;
        }

        if (newplayer == null) return;

        playerlist.addElement(newplayer);
        newplayer.addControllerListener(this);

        // send this player to player GUI
        playerWindow = new RTPPlayerWindow( newplayer, cname);
    }
}
```

See `RTPUtil` in "RTPUtil" on page 223 for a complete example.

IBM

Application Server Solution Guide Enterprise Edition: Getting Started

Barry Nusbaum, Bill Moore, Cristian E. Roldan
David Turner, Matthew Alcock, Steen Colliander



International Technical Support Organization

www.redbooks.ibm.com

SG24-5320-00

SG24-5320-00

IBM International Technical Support Organization

Application Server Solution Guide Enterprise Edition: Getting Started

May 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, “Special notices” on page 557.

First Edition (May 2000)

This edition applies to Versions 3.0 and 3.02 of WebSphere Application Server Enterprise Edition for use with Windows NT and AIX.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

Contents

Preface

This redbook will help you install, configure, and use WebSphere Application Server Enterprise Edition. We take an early look at the tools and technologies included in the Enterprise Edition package, with our focus on the Enterprise Java programming model, as opposed to the CORBA model that is also supported by the Enterprise Edition. The redbook, *WebSphere Application Server Enterprise Edition Component Broker 3.0 First Steps,* SG24-2033, discusses the use of the CORBA model.

This redbook gives a broad understanding of Enterprise Edition architecture and an introduction to the setup and use of some of its key components. It will be particularly useful to first-time users of Enterprise Edition. We assume that you have a good understanding of the Java language and some knowledge of Web-based application development.

Chapter 1, "WebSphere applications" on page 1 contains an introduction to WebSphere applications, including an overview of the key components of the Enterprise Java programming model, and what tools are used to edit, manage, deploy, run, and monitor applications.

Chapter 2, "WebSphere Application Server overview" on page 31 is a high-level overview of the architecture and functions of the Enterprise Java Server deployed in IBM WebSphere. There are also details on installation. More detail is given in Chapter 3, "Managing the infrastructure" on page 47 on how to manage servlets, Web applications, JSPs, servlet engines and EJBs in this environment.

Chapter 4, "Development tools" on page 87 is about how to use IBM-supplied development and test tools to build Java applications for deployment in WebSphere.

Chapter 5, "EJB deployment" on page 295 is a more detailed look at deployment of EJBs into WebSphere.

Chapter 6, "WebSphere interoperability" on page 337 considers how Enterprise Java applications in WebSphere can interact and operate with other Enterprise products that are part of the entire WebSphere Enterprise Edition. It includes information on interaction with TXSeries and Component Broker.

Chapter 7, "WebSphere access control and security" on page 363 looks at the WebSphere security model and how it interacts with other products including DCE, SecureWay Policy Director, and RACF. This is expanded

further in Chapter 8, "Directory Services" on page 403 where we look at Directory Services and how to configure WebSphere to use LDAP.

Chapter 9, "Debugging logging and tracing" on page 449 considers how to solve problems with the facilities provided by WebSphere, including debugging tools.

Finally in Chapter 10, "WebSphere 3.0 samples" on page 493 we look at the standard samples distributed with WebSphere and describe how to configure and run these samples.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

Barry Nusbaum is a Senior Technical Consultant at the Solution Partnership Center in Waltham, Massachusetts. He writes extensively and teaches IBM classes worldwide on all areas of UNIX and Windows systems management. He is also currently working on projects related to the UNIX operating system. Before joining the Solution Partnership Center, he worked in the ITSO for seven years. He can be reached at bnusbaum@us.ibm.com.

Bill Moore is a Senior Aim Consultant at the IBM Transarc lab in Sydney, Australia. He has 15 years of application development experience on a wide range of computing platforms and using many different coding languages. He holds a Master of Arts degree in English from the University of Waikato, in Hamilton, New Zealand. His current areas of expertise include the VisualAge family of application development tools, object-oriented programming and design, and e-business application development. He can be reached at billm@au1.ibm.com.

Cristian E. Roldan is a WebSphere Specialist in IBM Argentina. He has four years of experience in AIX and three years in Java and other development tools for e-business. He is also currently working on projects related to e-business and WebSphere for several platforms. He holds a degree from the University of Salvador at Buenos Aires, Argentina. He has worked at IBM for six years. His areas of expertise include WebSphere Application Server, VisualAge Java and AIX. He can be reached at roldanc@ar.ibm.com.

David Turner is a Consulting I/T Architect for the IBM Transarc Lab in Pittsburgh and is based in Austin, Texas. He has 20 years of experience in a wide variety of computing environments, including all IBM platforms and tools. He holds a Bachelors of Arts degree from the University of Texas at

Austin and is a certified TXSeries CICS monitor specialist. His areas of expertise include project planning and management, CICS, IMS, DB2, and distributed system design and deployment. You can reach him at dtturner@us.ibm.com.

Matthew Alcock is an IT Specialist at IBM Global Services Australia. He has over two years of experience in working with IBM software technologies. He holds a Bachelor of Science degree in Computing Science from the University of Technology, Sydney. His areas of expertise include AIX, Windows NT, DB2 UDB and WebSphere.

Steen Collander is a Senior IT Specialist in IBM Denmark. He has worked at IBM for 10 years. His areas of expertise include AIX and solutions within the e-business Application Framework with a focus on the IBM WebSphere product family and IBM Net.Commerce.

Thanks to the following IBM employees:

Nataraj Nagaratnam, WebSphere Security, Raleigh

Thomas Alcott, WebSphere Technical Sales Support, Costa Mesa

Ashok Iyengar, Customer Solutions Development, Pittsburgh

Melissa Pike, Customer Solutions Development, Pittsburgh

Inga Chapman, CICS Development, Hursley

Jenny Walls, Hursley

Wayne Winsor, CICS Support, Raleigh

Maggie Archibald, Performance Tuning, Dallas

Heinz Johner, DCE and Security, Austin

Larry Schick, Security Specialist, Los Angeles

Morgan Kinne, WebSphere Studio Development, Raleigh

Vincent Illiano, WebSphere Application Server Development, Raleigh

Sajith Kizhakkiniyil, WebSphere Studio Development, Mountain View

Comments welcome

Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in "IBM Redbooks review" on page 573 to the fax number shown on the form.
- Use the online evaluation form found at `http://www.redbooks.ibm.com/`
- Send your comments in an Internet note to `redbook@us.ibm.com`

Chapter 1. WebSphere applications

This chapter provides an overview of WebSphere applications. Throughout this chapter we take a look at what an application is, what components are used to form applications, and what is used to edit, manage, deploy, run and monitor applications.

1.1 What is an application?

An application in the context of WebSphere is a collection of user-supplied resources. Some examples of this are servlets, Enterprise Java beans (EJB), JavaServer Pages (JSP), static HTML, object groups and URLs.

An application is created so that a group of resources can be managed as a single entity, enabling users to easily specify conditions such as startup and exit dependencies over the entire group rather than over each component.

WebSphere's description of an application is an object that represents a collection of *live objects* with specific startup dependencies defined by the user. Applications are *top-level* objects, and as such, they cannot be contained by any other repository object.

The following section provides a brief overview of the major components used to make up an application. For further information refer to Chapter 3, "Managing the infrastructure" on page 47.

1.2 What is a servlet?

Servlets are small, server-side, platform-independent Java programs that enable a Web server to extend its capabilities with minimal overhead, maintenance and support. Servlets are compiled bytecode which can be dynamically loaded, or downloaded across a network, with no platform-specific considerations or modifications. This provides servlets with the capability of being written once and run anywhere.

Servlets are analogous to Java applets in the sense that the applets run on the client machine and the servlet, being a Java program like an applet, runs on the server. Just the fact that the servlets run on the server is a big advantage over applets because they have access to the server's resources. Servlets can then exercise a lot more control on what information should be sent out to the clients. In addition, there is no requirement for servlets to be graphically visible to the user, whereas, applets would be.

Servlets use a standard Java Application Programming Interface (API) as described in 1.2.2, "Java Servlet API V2.1 overview" on page 4 along with the associated classes and methods, which are supported by most major Web servers. Servlets can also use additional Java classes and packages to extend the Java Servlet API 2.1.

Servlets communicate through requests and responses, similar to the behavior of HyperText Transfer Protocol (HTTP). They interact with a servlet engine running on a Web server by using these requests and responses. For example, when a client sends a request to the server, the server can send the request information onto the servlet and have the servlet process the request, form a response which the server can then send back to the client.

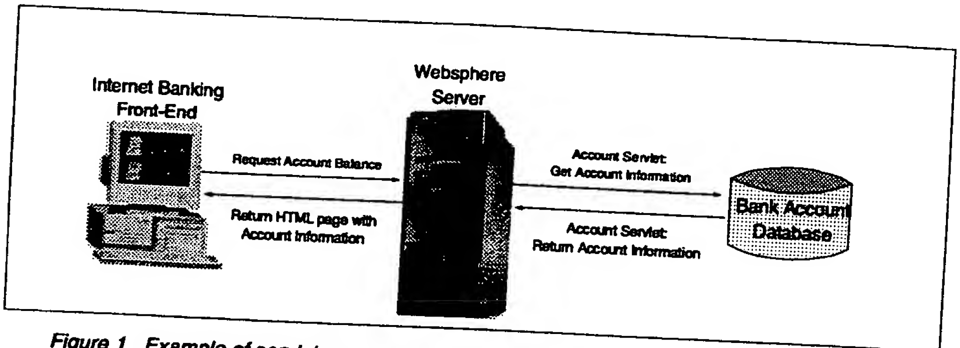


Figure 1. Example of servlet communication and interaction

Servlets can be loaded automatically when an applications is loaded, or they can be loaded the first time a client requests its services. Upon being loaded, a servlet will continue to run waiting for additional client requests. In addition, by using servlet aliases (servlet URLs), multiple instances are able to be created.

After the code for the servlet has been written, it has to be compiled, just like any other Java source code to generate the corresponding class file. To compile the servlet code, any Java tool/IDE can be used. After the code compiles without any errors and creates a class file, then we have to copy these files into appropriate directories. Other than the Java source code and the class file, we may have other associated JSP and html files. All these files need to go in the right folders before the servlet can be invoked.

1.2.1 Servlet lifecycle

Each servlet has the same lifecycle. It begins with the server engine loading the servlet into its memory and initializing it. The servlet is then ready to

handle requests from zero or more clients, until when the server removes the servlet.

1.2.1.1 Initializing a servlet

After the servlet is loaded into memory, the servlet engine attempts to create an instance of the servlet. This is usually done at the application startup, if that option is activated for the servlet, or at the first client request for the servlet after the application's startup.

The servlet engine creates an instance by creating the servlet configuration object and uses it to pass the servlet's initialization parameters to the *init* method. These parameters are applied to all invocations of the servlet until it is destroyed. The server may only load the servlet once. It must first destroy the servlet before reloading a new instance of the servlet.

If the initialization completes successfully, then the servlet is available to handle client requests. However, if the initialization fails then the servlet engine unloads the servlet. It is also possible that the administrator can set an application and its servlets to be unavailable for service. If this is the case the application and servlets will be unable to be run until the administrator changes their status to available.

1.2.1.2 Handling client requests

When a client request is received by the application server, the servlet engine creates a request object and a response object. These objects are then passed to the servlet *service* method when invoked by the servlet engine.

The service method gets information about the request from the request object. It then processes the request. It does this by invoking other methods such as doGet, doPost, doPut, doDelete or your own methods. It then uses the methods of the response object to pass back the response to the client.

1.2.1.3 Destroying a servlet

Servlets run until the servlet engine invokes the servlet's destroy method. This is usually caused by a request from the system administrator for an application to be stopped, causing the servlets belonging to that application to also be stopped.

Once the servlet is stopped the server then unloads the servlet. The Java Virtual Machine (JVM) performs a garbage collection sometime after the destroy.

1.2.2 Java Servlet API V2.1 overview

The Java Servlet Application Programming Interface (API) has been designed for today's and future request-response protocols. This expendability is provided by the API comprising two packages:

1. An HTTP-specific package
2. A non-HTTP-specific package

The IBM WebSphere Application Server Version 3.0 provides a servlet engine that implements the Java Servlet API V2.1. The application server implements the Java Servlet API V2.1 by including its packages in the application servers servlet.jar. The packages are:

- javax.servlet
- javax.servlet.http

These two packages contain seven interfaces, five classes and two exceptions. More information on these interfaces, classes and exceptions and the structure of a servlet can be found in the redbook *WebSphere Studio and VisualAge for Java - Servlet and JSP Programming,* SG24-5755-00. Also the JavaDoc documentation for the Java Servlet API V2.1 can be found at: `http://java.sun.com/products/servlet/2.1/api/packages.html`.

1.2.3 IBM extensions to the Servlet API V2.1

In its attempt to make it easier to manage session states and to create personalized Web pages, the IBM WebSphere Application Server Version 3.0 has included its own packages that extend and add to the Java Servlet API V2.1. The additional packages and classes are:

- com.ibm.websphere.servlet.personalization.sessiontracking package
- com.ibm.websphere.servlet.personalization.userprofile package
- com.ibm.websphere.db package
- com.ibm.websphere.servlet.error.ServletErrorReport class
- com.ibm.websphere.servlet.event package
- com.ibm.websphere.servlet.filter package
- com.ibm.websphere.servlet.request package
- com.ibm.websphere.servlet.response package

1.2.4 Servlet API V2.1 details

This section describes the functions of the Java Servlet API V2.1 as well as the IBM extensions to the API.

As listed in 1.2.3, "IBM extensions to the Servlet API V2.1" on page 4, the following packages and classes were added to extend the Java Servlet API V2.1 to make it easier to manage session state and to create personalized Web pages, and below are some details on what they actually provide.

com.ibm.websphere.servlet.personalization.sessiontracking package

This package provides the following:

- Records the referral page that led the visitor to your Web site.
- Tracks the visitor's position within the site.
- Associates user identification with the session.

com.ibm.websphere.servlet.personalization.userprofile package

This package provides the following:

- An interface for maintaining detailed information about visitors to your Web site. This is done by storing the information in a database.
- The ability to create Web applications that incorporate the detailed information, allowing you to create a personalized experience for each of your visitors.

com.ibm.websphere.db package

This package provides the following:

- Simplifies access to relational databases.
- Enhanced access functions, for example result caching, update through the cache, and query parameter support.

com.ibm.websphere.servlet.error.ServletErrorReport class

This class enables the application to provide more detailed and tailored messages when errors occur.

com.ibm.websphere.servlet.event package

This package provides the following:

- Listener interfaces for notifications of lifecycle events for applications and servers as well as servlet errors.

- An interface for registering listeners.

com.ibm.websphere.servlet.filter package

This package provides the following:

- Support for servlet chaining.
- The *ChainerServlet*, which needs to be added to an application to give it the ability to chain servlets.
- The *ServletChain* object.
- The *ChainResponse* object.

com.ibm.websphere.servlet.request package

This package provides the following:

- The *HttpServletRequestProxy* abstract class is used to overload the servlet engine's *HttpServletRequest* object, causing the overloaded request object to be forwarded to another servlet for processing.
- The *ServletInputStreamAdapter* class is used to convert an *InputStream* into a *ServletInputStream* and proxying all method calls to the underlying *InputStream*.

com.ibm.websphere.servlet.response package

This package provides the following:

- The *HttpServletResponseProxy* abstract class is used to overload the servlet engine's *HttpServletResponse* object, causing the overloaded response object to be forwarded to another servlet for processing.
- The *ServletOutputStreamAdapter* class is used to convert an *OutputStream* into a *ServletOutputStream* and proxying all method calls to the underlying *OutputStream*.
- The *StoredResponse* object is used to cache a servlet's response that contains data that is not expected to change for a period of time.

1.2.5 Changes to packages supported in WAS V2

The *com.ibm.servlet.connmgr* was used in the Application Server Version 2 to let a servlet communicate with a connection manager, that maintained a pool of open data server connections to JDBC or ODBC server products. This allowed the servlet to communicate directly with the data server using the connection manager's APIs when a connection was made from the pool.

This package has been deprecated, as the Application Server Version 3 now has a built-in connection pooling function. This now allows developers to write servlets to use the JDBC APIs to access the connection pool instead of using the connection manager's APIs directly.

Also in the Application Server Version 3 the following packages have been removed:

- *com.ibm.servlet.personalization.sam*
- *com.ibm.servlet.servlets.personalization.util*

1.3 What are Enterprise Java beans (EJB)?

This section will describe the Enterprise Java beans (EJBs) in the context of WebSphere V3.0. This section doesn't provide a detailed description of how to write EJBs. For that, the reader is encouraged to refer to other documents on that topic. A good source of reference on how to write EJBs in WebSphere is the IBM book *Writing Enterprise Beans in WebSphere*, SC09-4431-01. This book is shipped as part of the documentation with WebSphere V3.0. It can also be viewed or downloaded from:

`http://www.ibm.com/software/webservers/appserv/library.html`

1.3.1 Introduction

The EJBs are based on the component architecture. They are Java components which are used in a distributed client server environment. Typically, EJBs reside in the middle layer or the application layer (where the business logic resides) and provide the business logic implementation. These EJB components are lightweight, modular and easy to deploy. EJBs can be combined with other components to build an enterprise application. IBM's implementation of EJBs is based on the Sun Microsystems Enterprise JavaBeans specifications which can be found at:
`http://java.sun.com/products/ejb/docs.html`

There are two main categories of enterprise beans. They are shown in Figure 2 on page 8.

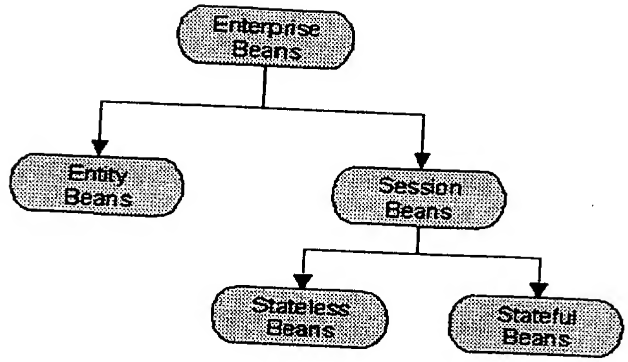


Figure 2. Categories of enterprise beans

1.3.1.1 Entity beans

Entity beans encapsulate permanent data and associated methods to manipulate the data. For their data to be permanent it needs to be stored somewhere, usually within a file system or a database such as IBM UDB DB2 or any other supported relational or object-oriented database. Instances of an entity bean are unique, but each bean can be accessed by multiple users.

The data can be synchronized in two ways. When the bean handles its own data synchronization, the process is called bean managed persistence (BMP). On the other hand when the container handles the data synchronization, the process is called container managed persistence (CMP). The entity bean lifecycle is shown in Figure 3 on page 9.

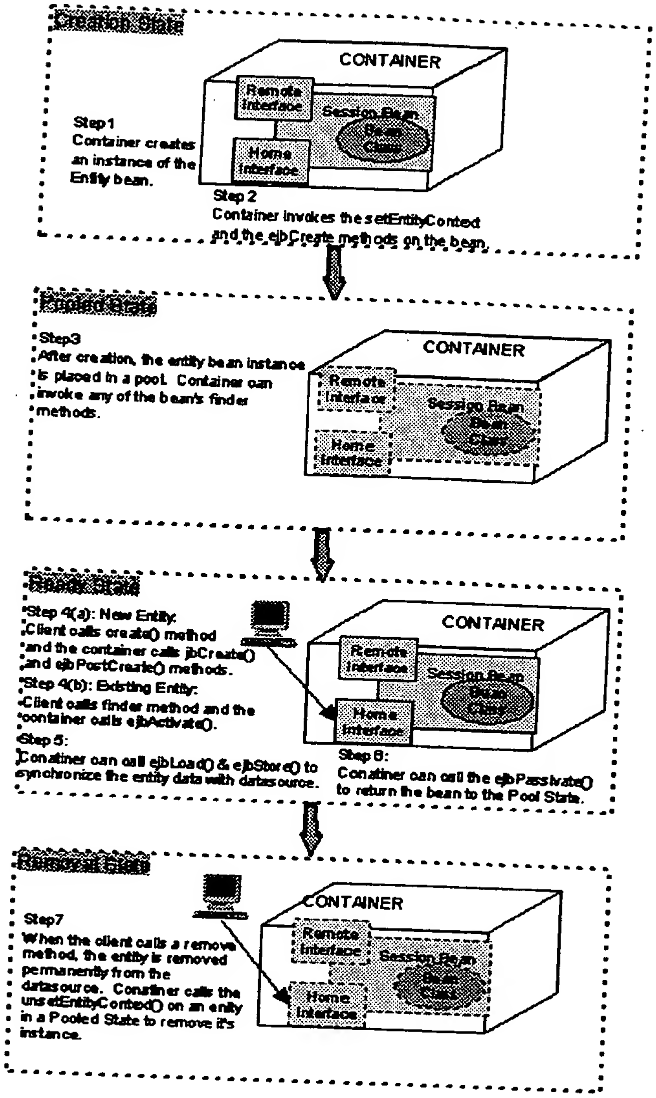


Figure 3. Entity bean life cycle

Each entity bean consists of the following 4 components:

a. Bean Class: This class is implemented by the developer to encapsulate the business methods and data. This class is hidden from the clients

and is accessed by the remote interfaces implemented by the container.

b. Home Interface: Container implements this interface and provides methods to create, find and remove the instances of the enterprise bean.

c. Remote Interface: This is the other interface that is implemented by the container when the enterprise bean is deployed in a container. The remote interface provides the clients access to the business data and methods in the bean class.

d. Primary Key Class: Since entity bean instances are unique, this class encapsulates one or more variables and methods to manipulate those variables, which are used as a primary key to uniquely identify a bean instance.

1.3.1.2 Session beans

Session beans encapsulate non-permanent data. They perform units of work on behalf of an EJB client. Typically, the session beans lifetime is that of the EJB client it is servicing. Unlike the entity beans, their data is not required to be stored in a data source. Session beans can however, make this data persistent by using underlying entity beans.

Every session bean has the following three components:

- Bean class
- Home interface
- Remote interface

The attributes of these interfaces and classes are the same as those of the entity beans. The reader will observe that the session bean does not have a Primary Key class as does the entity bean. The reason for this is that unlike the unique entity bean instances, the session bean instances are not unique. These instances are identical. When session beans encapsulate any semi-permanent data, then their instances become unique.

Each session bean is associated with one client. A session bean lifecycle is outlined in Figure 4 on page 11. Depending on the life span of a session bean they can be further classified as Stateful or Stateless session beans. A Stateful session bean maintains data across methods and thus has a longer life-span. On the other hand a Stateless session bean does not maintain data across the methods and is short-lived.

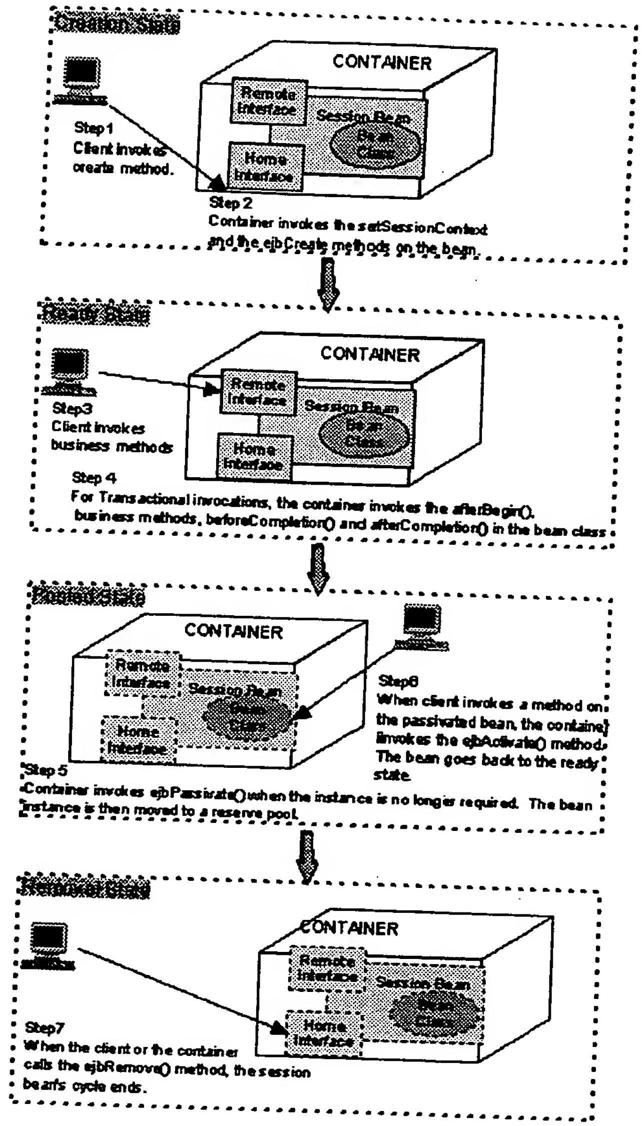


Figure 4. Session bean life cycle

1.3.2 EJB architecture in brief

The EJB architecture is based on the Sun Microsystems Enterprise JavaBeans specification. The EJB environment allows the users of the WebSphere Advanced and the WebSphere Enterprise editions to integrate their Web-based systems with their other business systems.

The EJB implementation consists of four major components. These are shown in Figure 5 on page 13.

1. EJB server

   In a 3-tier architecture scheme, the EJB server is the application server layer. As shown in Figure 5 on page 13, the EJB server connects the clients (for example servlets, JSPs or Java applications) to the enterprise data. The EJB server contains the business logic in the form of Java beans. These Java beans are deployed in a container, and a container is deployed in the EJB server. The clients cannot access the business logic and the business data in the Java beans directly. They have to go through the remote interface implemented by the container. The Advanced edition of WebSphere ships with only one EJB server called the EJB server (AE). The Enterprise edition of WebSphere ships with two EJB servers namely - the EJB server (AE) and the EJB server (CB). The EJB server (AE) includes only one standard container that supports both the session and the entity beans. The EJB server (CB) has two containers - the entity container for the entity beans, and the session container to hold the session beans.

   The reader can find detailed information on EJB servers in *Writing Enterprise Beans in WebSphere*, SC09-4431-01, that ships with WebSphere V3.

   This book can also be viewed or downloaded from:

   `http://www.ibm.com/software/webservers/appserv/library.html`

2. Data source

   The persistent data in the entity beans is stored in a recoverable data source. For container managed persistence (CMP), the EJB server (AE) supports IBM DB2, Oracle, and Microsoft SQL Server, and the EJB server (CB) supports DB2, Oracle, IBM CICS and IBM IMS.

   For bean managed persistence (BMP), the user can use any data source or a file stem to store the persistent data. The user will then have to write code for the beans to handle their own data source interactions.

3. EJB clients

The EJB clients can be one or more of the following: Java servlet, Java applet-servlet combination, or a JSP file. A Java applet can be used with a servlet to interact with the enterprise beans, while in the EJB server (CB), a Java applet can directly interact with the Enterprise Java beans. In an EJB server (CB) environment, additional EJB clients can be ActiveX clients, a CORBA-based Java clients, and to a limited degree a C++ CORBA clients. More details on how to write EJB clients can be obtained by the reader from *Writing Enterprise Beans in WebSphere*, SC09-4431-01.

4. Administration interface

   The EJB server (AE) uses the WebSphere Administrative Console to administer the EJB server. The EJB server (CB) uses the System Management End User Interface (SMEUI).

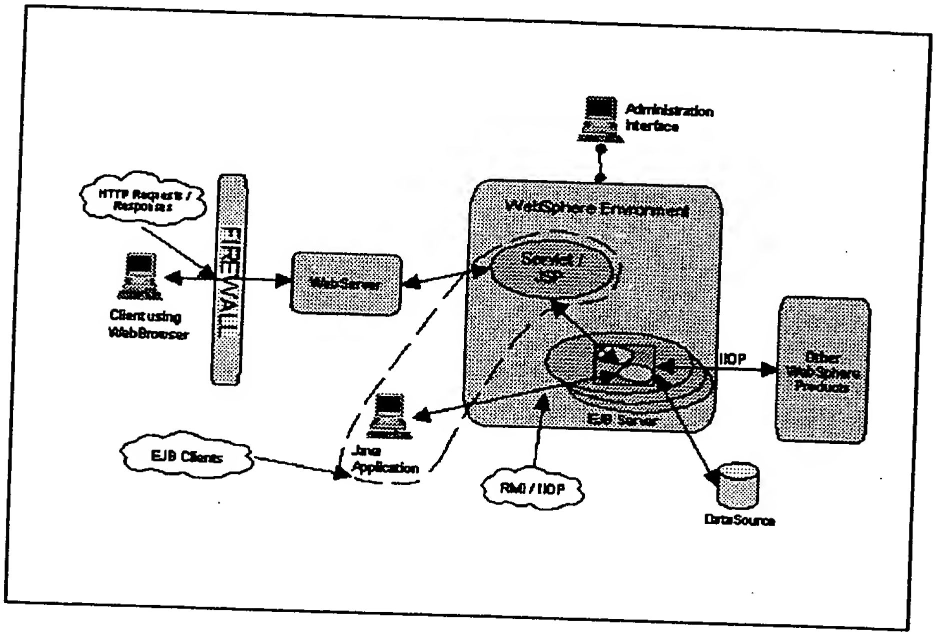


Figure 5. EJB environment and interaction with other components

1.4 What are JavaServer Pages (JSP)

JavaServer Pages (JSP) technology provides developers with an easy and powerful way to build Web pages with dynamic content. JSPs dynamically generate HTML, eXtensible Markup Language (XML), and other structured

documents inside a server, and enable you to effectively separate the structured documents from the business logic in your Web pages.

The IBM WebSphere Application Server programming model implements the JavaSoft JSP Specification. In addition IBM has added to the JSP specifications with JSP tags that are HTML-like which will make it easier for HTML authors to add the power of Java to their Web pages. The Application server provides support for two levels of the JSP specifications, which are JSP 1.0 and JSP 0.91.

JSP technology-enabled pages share the same ability as servlets, in that they are written once, but can be run anywhere.

1.4.1 JSP process flow

The JSP process flow is shown in Figure 6 on page 15.

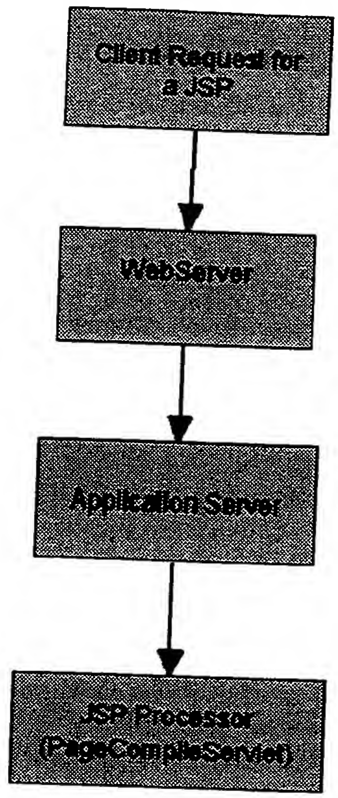


Figure 6. JSP process flow

When WebSphere is installed on top of a Web server, the Web server's configuration is modified to pass the HTTP requests for the JSP files to the WebSphere application server. WebSphere then passes this request to the JSP processor which is nothing but a Java servlet which compiles the JSP file and in the process creates two files for each JSP file. It creates a .java file which has the Java code for the servlet, and a .class file, which is a compiled bytecode for the .java file. The JSP Processor for JSP 0.91 is the

com.ibm.servlet.jsp.http.pagecompile.PageCompileServlet

and for JSP 1.0 it is the

com.sun.jsp.runtime.JspServlet

The JSP processors puts the .java and the .class files inside the \WebSphere\AppServer\Temp folder. The location of these files will depend on whether you are using JSP 0.91 or JSP 1.0 in your application.

For example, if you are using JSP 0.91 and the JSP file is in the following folder:

<WASRoot>\hosts\default_host\examples\web, then the JSP processor will place the .java and the .class files in the following path:

<WASRoot>\temp\examples\pagecompile folder.

The JSP 0.91 processor uses a naming convention to name these .java and the .class files. If, for example, the JSP filename is simple.jsp, then the processor will name the .java and the .class files as _simple_xjsp.java and _simple_xjsp.class, respectively. Under JSP 0.91 the .java file is always kept.

If you are using JSP 1.0 and the JSP file is in the following folder:

<WASRoot>\hosts\default_host\examples\web, then the JSP processor will place the .class files in the following path:

<WASRoot>\temp\examples

The JSP 1.0 processor names the .class file simple.class and by default the .java file is not kept after compilation. To keep the .java file it is necessary to set the Init Parm Name `keepgenerated` used by the JspServlet to `true`. You can do this from the WebSphere Administrative console as shown in Figure 7 on page 17

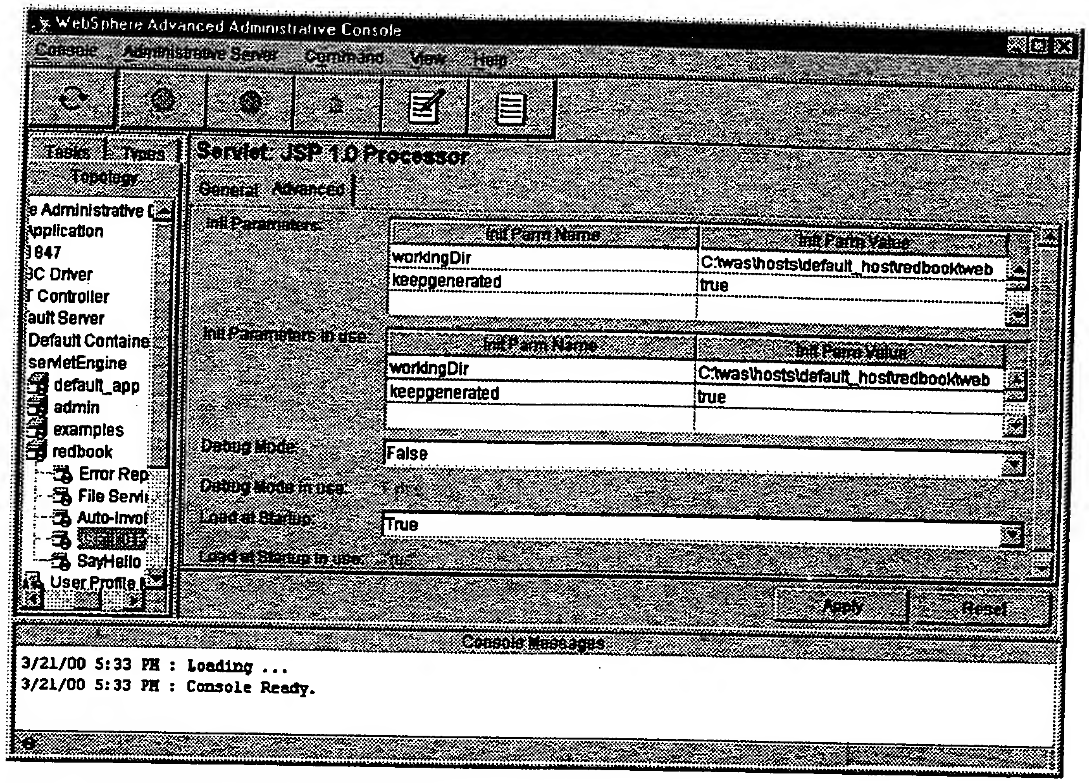


Figure 7. Setting Init Parm Name for JspServlet

The .java and the .class files generated from the JSP file are servlets extending the javax.servlet.http.HttpServlet class. A sample JSP 0.91 file and the Java code generated by it is shown in Figure 8 on page 18.

```
<BEAN name="simpleSessionMsg" type="SimpleJSPBean" create="true"
scope="session"></BEAN>
<BEAN name="simpleRequestMsg" type="SimpleJSPBean" create="true"
scope="request"></BEAN>
<%
   SimpleJSPBean simpleApplicationMsg =
   (SimpleJSPBean)
getServletContext().getAttribute("simpleApplicationMsg");
%>


<html>
<head><title>Simple JSP</title></head>
<body>
```

```
<img src="banner.gif" alt="banner image" height=100 width=584><p>
<h1>Simple JSP page</h1>
<% if(simpleApplicationMsg != null){ %>
<B>Application Bean:</B>  <%=simpleApplicationMsg.getMessage() %><BR>
<% } %>

<B>Session Bean:</B> <%= simpleSessionMsg.getMessage() %> <BR>

<B>Request Bean:</B> <%= simpleRequestMsg.getMessage() %> <BR>

</body>
</html>
```

Figure 8. simple.JSP code 0.91

When this file is processed by the JSP processor, the _simple_xjsp.java file is generated. In Figure 9 on page 19, only a snippet of code is shown. Please refer to Appendix B, "Sample Code" on page 549, for all of the code and an example of the code generated by the JSP 1.0 compiler.

```
package pagecompile;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.beans.Beans;
import com.ibm.servlet.jsp.http.pagecompile.ParamsHttpServletRequest;
import com.ibm.servlet.jsp.http.pagecompile.ServletUtil;
import com.ibm.servlet.jsp.http.pagecompile.filecache.CharFileData;
import com.ibm.servlet.jsp.http.pagecompile.NCSAUtil;

import SimpleJSPBean;

public class _simple_xjsp extends javax.servlet.http.HttpServlet {
    private static final String sources[] = new String[] {
        "c:\\websphere\\appserver\\hosts\\default_host\\examples\\
web\\simple.jsp",
    };
    private static final long lastModified[] = {
        926708647000L,
    };

    public void service(HttpServletRequest request,
HttpServletResponse response)
```

```
            throws IOException, ServletException
        {
    -------------
    -------------
        }

            SimpleJSPBean simpleSessionMsg= (SimpleJSPBean)
                request.getAttribute("simpleSessionMsg");
            if ( simpleSessionMsg == null )
                throw new ServletException("Invalid BEAN name:
simpleSessionMsg");
            {
            java.util.Properties p = new java.util.Properties();
            java.util.Enumeration e = request.getParameterNames();
            while (e.hasMoreElements()) {
                String name = (String) e.nextElement();
                p.put(name, request.getParameter(name));
            }
            com.ibm.servlet.util.BeansUtil.setProperties(simpleSessionMsg, p);
            }
------------------
------------------
}
```

Figure 9. Code snippet of the _simple_xjsp.java file generated by the JSP processor

1.4.2 JSP lifecycle

Since after compilation, JSPs generate a servlet, their life cycle is similar to that of a servlet. When the ServletEngine receives a request for a JSP file, it checks to see if the servlet already exists or if the JSP file has changed since the last time it was invoked. If the servlet for the JSP does not exist in the application classpath or if the JSP file was changed since the last time it was loaded, the servlet engine passes the request to the JSP processor (or the pagecompile servlet). This creates another .java and .class file for the requested JSP file. These files are placed in the application classpath. The servletengine then creates an instance of the class file and calls the servlet service() method in response to the request. Once the .class and .java files have been created by the JSP processor, all the subsequent requests for the JSP servlet are handled by the instance of the servlet that was created by the servletengine. By default, the JSP syntax in a JSP file is converted to Java code by the processor and this code is placed in the service() method of the generated class file. This default behavior can be overridden by using the method directive in the JSP file.

The JSP servlet is terminated when the servletengine no longer needs the servlet or a new instance of the servlet is being loaded by the servletengine.

In doing so, the destroy() method in the servlet is called. If there is a need to conserve resources or if the previous request to the servlet times out, then also, the servletengine can call the destroy() method of the servlet.

1.4.3 JSP access models

When using JSPs there are commonly two types of access models used:

1. JSP file handles both the request and the response

   In this model the JSP file handles both the request and the response to and from the client browser. The JSP passes the request to beans or other components to generate the dynamic content. The response is sent back to the JSP file from the beans, which in turn sends it back to the client browser.

2. JSP handles response, servlet handles request

   In this model, the client request is sent to the servlet, which handles the dynamic content generation. It then calls a JSP file to send the response back to the client. Using this model, one can effectively separate the business logic from the content display.

These two JSP access models are discussed in more detail in the redbook *Patterns for e-business:User to Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition,* SG24-5864-00.

1.4.4 JSP syntax

In WebSphere V3, JSP conforms to the JavaServer Pages Specifications 0.91 or 1.0 from Sun Microsystems. IBM has added some enhancements particularly in the area of database access. The Sun JSP 1.0 specification can be found at:

```
http://java.sun.com/products/jsp/download.html
```

A full discussion of JSP syntax including the differences between Version 0.91 and 1.0, and details of the IBM extensions can be found in the redbook *WebSphere Studio and VisualAge for Java Servlet and JSP Programming,* SG24-5755-00

1.5 Enterprise Java Server (EJS) Runtime

The Enterprise Java Server (EJS) Runtime provides support for the Enterprise Java beans (EJB) programming model in which enterprise beans are managed by containers. Containers, in turn are executed within servers which are operating system processes that contain their own Java Virtual

Machine (JVM). Each JVM is managed by the System Management (SM) infrastructure of the application server.

The SM infrastructure allows the execution environment to be defined, enabled and monitored. The execution environment is made up of beans, containers and servers.

For more detailed descriptions and examples on each of the EJS functions please refer to 2.1, "IBM WebSphere Application Server components" on page 31.

1.5.1 Enhancements to the IBM extensions required for the EJS

The EJS programing model utilizes the RMI/IIOP model to provide distribution for the EJBs in standard and advanced releases of the EJS. The enterprise release of EJS requires the use of Interface Definition Language (IDL) to allow interoperability with the Component Broker (CB).

Some of the existing IBM extensions in the IBM Java ORB have been re-implemented as an effort by the Component Broker team to enhance the JBroker Java ORB to support the EJS. Some of these features are briefly discussed below.

1.5.1.1 Persistent Name Service (PNS)

PNS is not an ORB feature but it is required to provide reference to the persistent objects. PNS implements CORBA CosNaming specification and this mechanism will be standardized for pluggable persistence.

1.5.1.2 Object Resolver

The Object Resolver provides a pluggable interface to allow an external class to act as a specialized object adapter.

1.5.1.3 Request Interceptor (RI)

The RI allows access to the request header after marshalling out and before demarshalling in. This was done because the Request and Message Interceptor design from the original Sun ORB was not compatible with how C++ ORB handled interceptors.

1.5.1.4 Property setting and getting flags

Since the OMG defines only a couple of basic properties (`org.omg.CORBA.ORBClass`, `org.omg.CORBA.ORBSingletonClass`), they are not enough for IBM-specific properties. Support for these properties has been added and it will affect the method `ORB.set_parameters` and other affiliated methods.

1.5.1.5 JNDI support

EJS implements the Java JNDI APIs to support the CORBA CosNaming.

1.5.1.6 Java Transaction Service (JTS)

JTS supports the ORB. This is done by implementing the Request Interceptors, which allows the JTS to be enabled by the ORB.

1.5.1.7 Browser callbacks

Normally, a client makes method calls to an object residing on the server. This is done when the server creates a listener socket and the client opens a port to that socket. By using the same listener socket, created by the server, and the port, opened by the client, functionality has been added for the server to make method calls to the object residing in the client. The roles have been essentially reversed. This is possible only in the case of signed applets. A similar capability has been added to the C++ ORB.

1.5.2 New features in Java ORB required for the EJS

Some of the new features in the IBM Java ORB, that are required by the EJS runtime have been briefly discussed below.

1.5.2.1 Pluggable feature framework

In WebSphere V3.0, special emphasis has been paid to providing a very pluggable component-based framework. This kind of framework will take care of problems arising from receiving frequent updates of the Java ORB from Sun Microsystems and it will meet the needs of different internal customers with varying requirements. This feature is meant for private use only by the ORB team

1.5.2.2 Pluggable transport layer

In order to support any environment, it is necessary to have a pluggable transport layer. EJS has a pluggable transport layer, which means that the socket creation will have to take place in this layer. Other ramifications are that the CDRInputStream and CDROutputStream will also have to be replaceable.

1.5.2.3 SSL support

Support for the SSL interface on the server side has been provided. Using the MIME encoded IIOP allows browsers and Web servers to use SSL. By selecting to enable SSL, the user can force an SSL connection between the client and Web server for greater protection of the user ID and password data.

1.5.2.4 Mime-encoded IIOP tunneling with HTTP

IIOP tunneling means sending IIOP messages embedded in an HTTP request/response. When the client or a firewall does not allow a post method, the mime encoded IIOP tunneling technique can be used. This technique allows the mime encoded IIOP to tunnel through an HTTP firewall or Web server as an HTTP message.

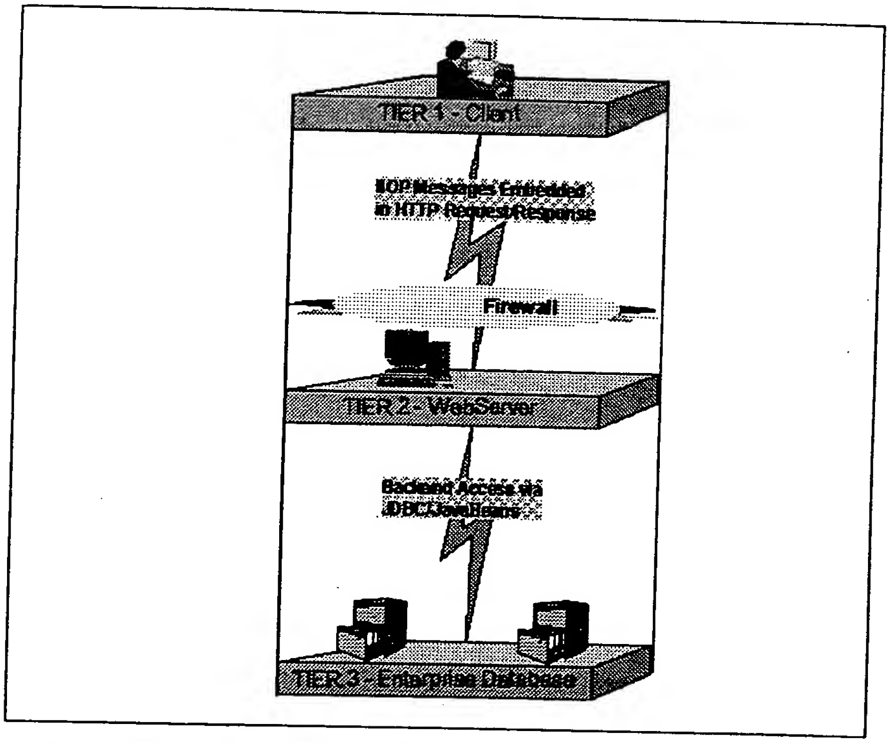


Figure 10. Mime encoded IIOP tunneling through a 3-tier framework

1.5.2.5 IIOP tunneling

There are 2 types of IIOP tunneling. In the first approach, multiple IIOP messages are embedded in an HTTP request and are communicated to the ORB using sockets. This approach has some security implications on Web servers and will not be used. The second approach is to attach the IIOP request to a single HTTP request as a parameter with binary data. This request reaches the ORB using the servlet that is started by the HTTP request. The response is then sent back as binary IIOP data. This approach is used since it is more secure.

1.5.2.6 IIOP firewall support

SOCKS V5 is used to provide the IIOP firewall support by providing authenticated connections.

1.5.2.7 IIOP redirector

IIOP redirector works in a similar fashion to the proxy firewall. It copies any replies/requests on a socket to a socket on which the ORB is connected.

1.5.2.8 Configurable requests time-outs

Only client-side timeouts have been implemented. This is done by throwing the NO_RESPONSE system exception on the client side with the completion status of *maybe*. The timeouts are set by using two ORB properties:

1. com.ibm.CORBA.RequestTimeout
2. com.ibm.CORBA.LocateRequestTimeout

The default value for both of these timeouts is 0. A timeout value of 0 means an infinite timeout interval.

1.5.2.9 eNetwork Dispatcher

The eNetwork Dispatcher is a WebSphere HTTP sprayer used from Work Load Management (WLM).

1.5.2.10 LDAP naming service

EJS provides the LDAP naming service support indirectly through a JDBC layer implemented for CosNaming. As part of LDAP support, the EJS also supports the URL context.

1.5.2.11 Work Load Management support (WLM)

WLM distributes the workload or requests across a server group. This functionality is supported in WebSphere V3. Smart proxies are used within the client along with a WLM runtime.

1.6 Preparing for Installation - What to change and why

Before running the setup of WebSphere for version 3.0 it was necessary to make some of the following changes, to ensure a good install. These hints were documented in the WebSphere Readme file that shipped with the product

1.6.1 Set the JAVA_HOME environment variable

Setting this variable allowed the OLT install to find the correct files as discussed in 4.3.0.2, "Installing and configuring the OLT/OLD environment" on page 268. The 3.02 WebSphere install on NT completes successfully even if the java_home variable is not set.

For the Windows NT platform:

1. Open the Control Panel.
2. Double click the **System** icon.
3. Select the **Environment** tab.

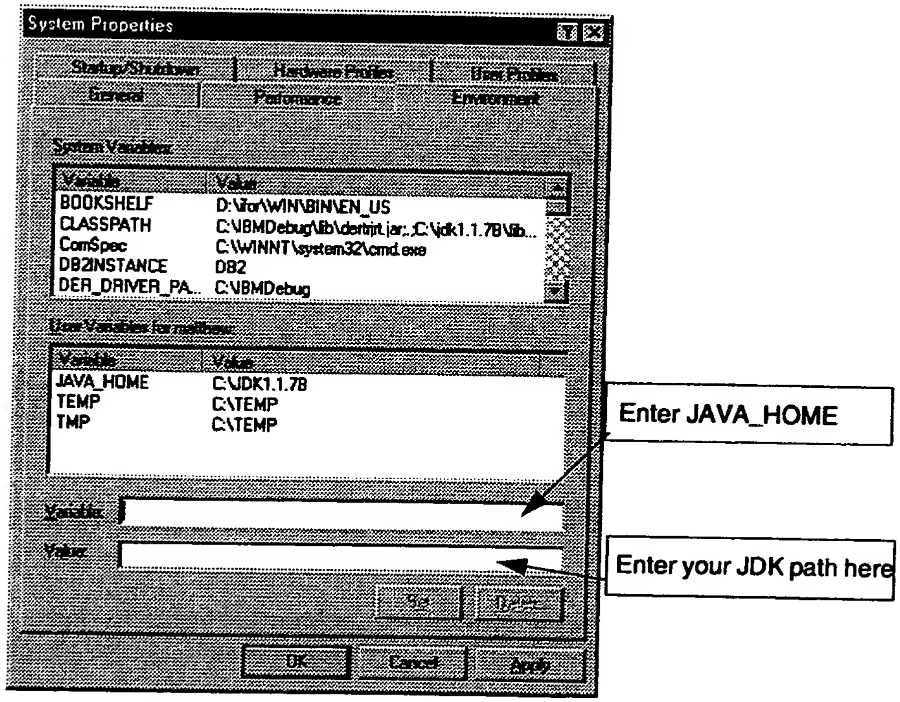


Figure 11. The Windows NT System Properties Environment settings

4. Type in JAVA_HOME in the variable field.
5. Type in your JDK path, for example `C:\JDK1.1.7B` in the value field.
6. Click **Set.**
7. Click **OK.**

For the AIX Platform:

1. Log in as root or a super user.
2. cd /etc
3. Edit the environment file with an editor of choice, for example, vi.
4. Add the line `JAVA_HOME=/usr/jdk_base`.
5. Save the file and exit. Next time you log in the environment will be updated. Below is an example of our settings within the /etc/environment file:

```
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin
TZ=EST5EDT
LANG=en_US
LOCPATH=/usr/lib/nls/loc
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
LC__FASTMSG=true
# ODM routines use ODMDIR to determine which objects to operate on
# the default is /etc/objrepos - this is where the device objects
# reside, which are required for hardware configuration
ODMDIR=/etc/objrepos
# IMNSearch DBCS environment variables
IMQCONFIGSRV=/etc/IMNSearch
IMQCONFIGCL=/etc/IMNSearch/dbcshelp
# Added by Steen
JAVA_HOME=/usr/jdk_base
LD_LIBRARY_PATH=/usr/jdk_base/lib/aix/native_threads:/usr/WebSphere/AppServer/pl
ugins/aix:/home/db2inst1/sqllib/lib:/usr/lib
```

Figure 12. /etc/environment settings

1.6.2 Increase the DB2 application heap size for the WAS database

In order for the Admin Server to work correctly the application heap size must be changed from its default setting of 128 to a new value of 256. In the 3.02 install on NT the installation will update this setting automatically for you. This can be done manually either through the DB2 UDB Control Center or through the db2 command line interface, below are examples of both.

From the DB2 UDB Control Center:

1. Open the Control Center and expand the system that the WAS Database resides on, so you can see the WAS DB on your screen, as shown below in Figure 13 on page 27:

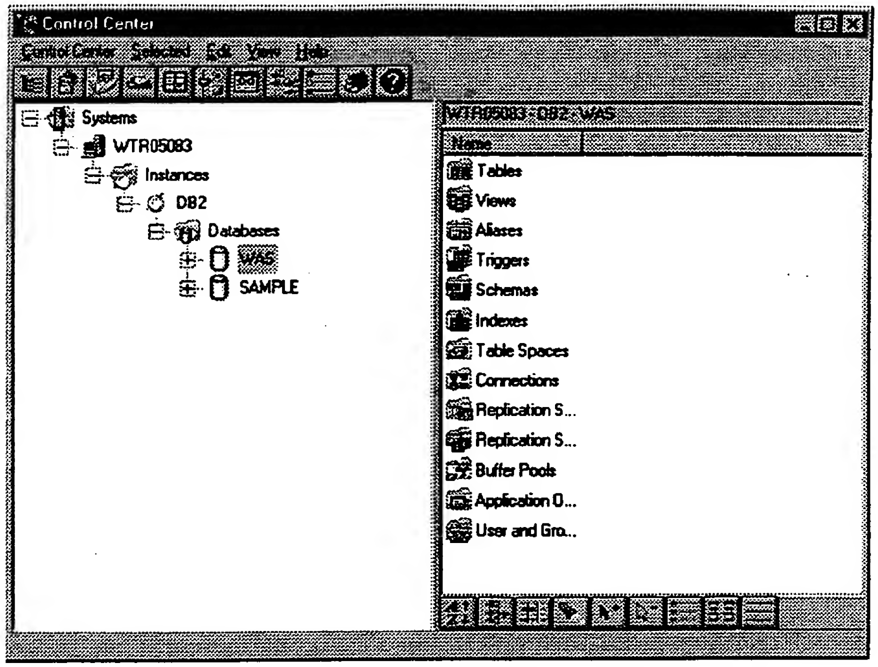


Figure 13. DB2 UDB Control Center - the WAS database

2. Right-click the **WAS** database icon and select **Configure**.

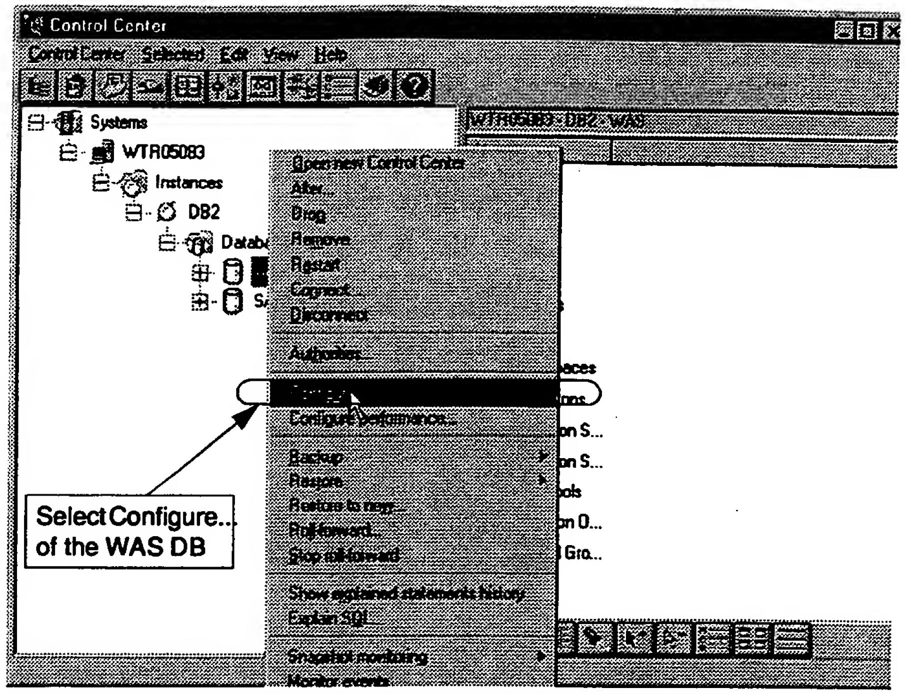


Figure 14. Configure the WAS database

3. Select the **Performance** tab.
4. Scroll down the list of parameters and select **Application heap size**.
5. Change the value from 128 to 256.

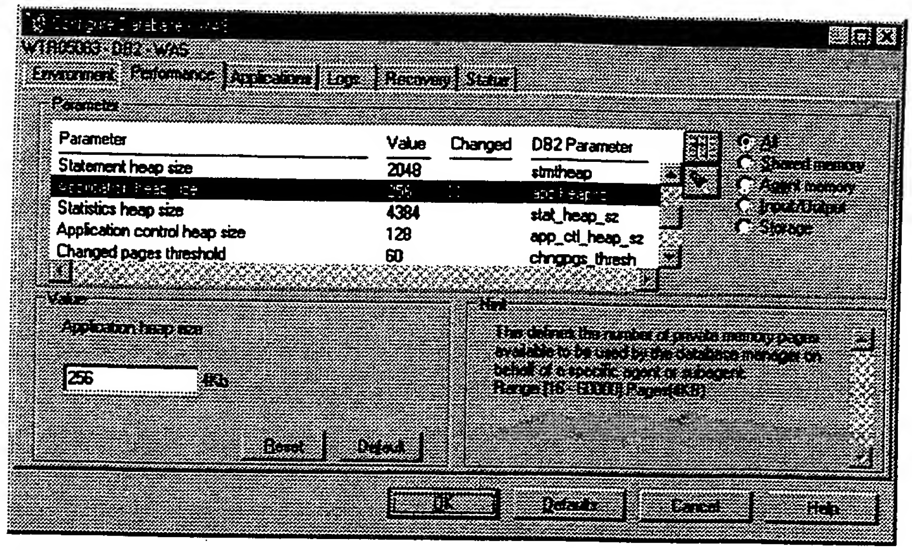


Figure 15. How to change the application heap size

6. Click **OK**.
7. Close the Control Center.

From the command line:

1. Start up a DB2 Command Line window (NT) or ensure your ~/sqllib/db2profile has been activated (AIX).
2. Connect to the WAS database with a valid user ID and password:

   ```
   connect to was
   ```

3. To update, type the following command:

   ```
   update db cfg for was using APPLHEAPSZ 256
   ```

4. Disconnect all applications:

   ```
   force applications all
   terminate
   ```

5. Then stop and start db2.

Chapter 2. WebSphere Application Server overview

This chapter will give a high-level overview of the Enterprise Java Server functions as deployed in IBM WebSphere V 3.0.

2.1 IBM WebSphere Application Server components

WebSphere Enterprise edition ships with two different Enterprise Java Servers as discussed in 1.3.2, "EJB architecture in brief" on page 12. The following is an overview of the main components in the EJS provided by WebSphere Advanced Edition (AE) and a brief look at the main components of the EJS provided by Component Broker (CB). We also discuss the features of WebSphere Standard Edition because they are included in Advanced Edition and they provide the necessary foundation for Advanced Edition functions.

2.1.1 Architecture overview

This section gives an overview of the components in the general EJS architecture.

Enterprise Java Services (EJS) refers to the infrastructure designed to run servlets and Enterprise Java beans.

The EJS is based on Sun's Enterprise JavaBeans Technology specifications that specify an enterprise Java platform defined through a set of standard Java APIs that provide access to existing infrastructure services.

2.1.1.1 EJS architecture

An EJB server provides the following components:

- The EJB server runtime

  The server runtime can be seen as a generic server (or model/template server) from which the Web application server instances can be modeled. EJBs live in containers that again live in the server runtime (Web application server) see Figure 18 on page 37.

  Servlets also live in a special container (servlet engine) that again live in the server runtime (Web application server) see Figure 17 on page 36.

- The EJB containers

  EJB containers are provided following the requirements as described in the Enterprise JavaBeans specifications Version 1.0 (for further information see `http://java.sun.com/products/ejb/`). Furthermore, IBM

provides additional features for example, entity bean support with bean managed or container managed persistency and a simple deployment tool.

See 2.1.2, "Enterprise Java beans and containers" on page 39 for more details on containers and EJBs.

- The Enterprise Java beans

This combination of components provides a number of services. These are:

- A deployment tool

  When an EJB has been developed it has to be transformed into a form that enables it to be managed and accessed. The transformation is referred to as EJB deployment.

  To deploy an EJB you will normally use a built-in tool of an IDE (Integrated Development Environment) or a stand alone tool.

  See 2.1.4, "Deployment tools" on page 40 for more details.

- Naming services

  The IBM WebSphere Application server architecture provides naming and directory services to provide an interface to find EJBs based on the name or an attached attribute.

  In IBM WebSphere the Java Naming and Directory Interface (JNDI) is used to provide a common interface to the actual naming and directory service that is being used.

  JNDI provides an Application Program Interface (API) to be accessed through a Java application and a Service Provider Interface (SPI) to specify the interface to existing and widely used name and/or directory services. The purpose of providing an open SPI specification is to make the JNDI independent of the specific naming or directory service implementation used.

  For further information on the JNDI specifications see:
  `http://java.sun.com/products/jndi/`

  For more details on naming and directory services see Chapter 7, "WebSphere access control and security" on page 363.

- Security services

  The security services provide support for Web resources (for example, HTML, JSP and CGI files), servlets and EJBs.

  The security authorization information, authentication and delegation policies will typically be defined using the IBM WebSphere Administrative

Console. In WebSphere Advanced the security service is an EJB server that contains security beans.

See Chapter 7, "WebSphere access control and security" on page 363 for more details on security.

- Work Load Management

  A Work load management (WLM) mechanism is provided to make scaling of enterprise applications running on IBM WebSphere Application server possible.

  Workload Management is a service that improves the scalability of an EJB server runtime environment by grouping multiple servers together into a server group. EJB clients can access this server group as if it was a single server. The actual server that responds to the client request will be transparently determined by the Workload Management service. WebSphere implements a number of different policies for how the Workload Management service will choose the server.

  We do not cover Workload Management in this book.

- A persistence service

  This service provides support for the proper interaction between a bean and its data source to ensure that any persistent data is maintained.

  In AE this is accomplished by using the JDBC API to interface with relational databases and Java transaction support.

  In CB the persistent service is accomplished using the X/Open XA interface to relational databases and the OMG Object Transaction service.

- A transaction service

  This service implements the transactional attributes specified in an EJB's deployment descriptor.

- System management infrastructure

  The system management infrastructure enables management of Web server, Web application server and Web application resources. A client interface is provided for the administrator to manage these resources.

  An overview of the AE system management infrastructure is given in 2.1.5, "System Management Infrastructure" on page 40.

  The IBM WebSphere Administrative Console is provided with IBM WebSphere 3.0 Advanced.

  See 2.1.5.1, "Systems management console" on page 41.

The client interface for the EJB server (CB) environment is the systems management end user interface. We do not discuss this interface in this redbook. Please refer to the redbook *WebSphere Application Server Enterprise Edition Component Broker 3.0 First Steps,* SG242033 for more details.

2.1.1.2 WebSphere Application Server Standard Edition

The WebSphere Application Server Standard Edition provides a basic Web application server environment for Web applications that typically consist of HTML files, JSP files, applets, servlets, image files and databases.

The primary purpose of the IBM WebSphere Application Servers is to provide an environment into which scalable, portable, well performing and reliable Web applications can be developed and executed based on Java-based programming, standard techniques, Internet standards, standard middleware and database management systems.

IBM WebSphere Application Server Standard Edition provides an environment where you can extend and enhance your Web applications by migrating your HTML to JSP. Since JSPs essentially are HTML files with additional JSP-specific tags you can use your HTML skills and standard development tools.

IBM WebSphere Application Server compiles the JSPs (at runtime) and transfers them into servlets. However, this process is managed by IBM WebSphere Application Server and is transparent to the developer.

Since JSPs can include Java code and you write your servlets in Java you can use Java in your development - if you want to or the requirements force you to. However, you are not necessarily required to do so.

"Traditional" Web development components like CGI programs, Web Server APIs and client side scripting languages may also be utilized since an IBM Web Application Server setup includes a "traditional" Web server.

IBM WebSphere Application Server Version 3 integrates with many different Web Servers and includes plug-ins for IBM HTTP Server, Apache, Lotus Domino Go V4.6.2.5, Lotus Domino, Netscape Enterprise Server, and Microsoft IIS.

To integrate with a Web server you must select one or more of the plug-in extensions as shown in Figure 16 on page 35.

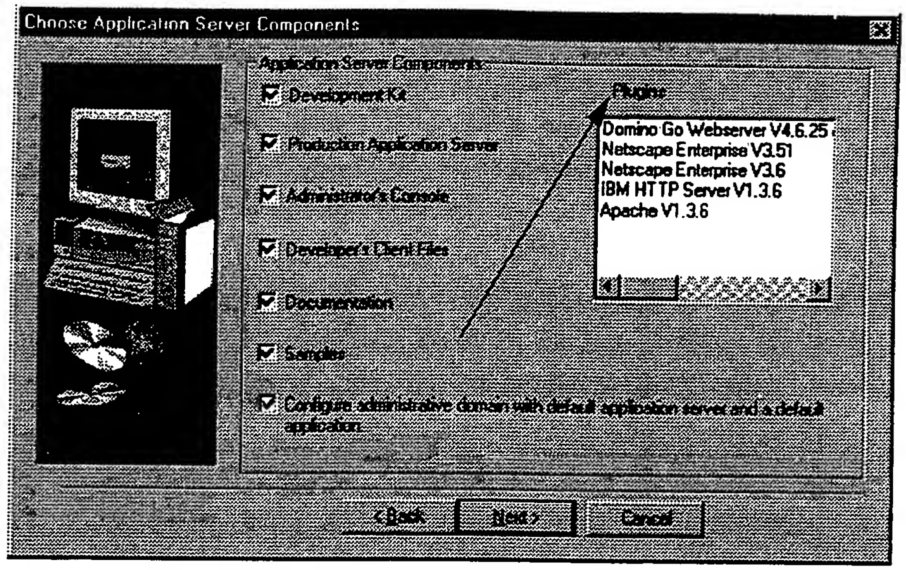


Figure 16. Web servers that integrate with the application server

The application server provides the runtime environment for execution of servlets. See Figure 17 on page 36 for a view of this runtime architecture.

The Web application server also provides a connection manager function that manages database connections. The connection manager provides an easy to use mechanism for reducing the resources required by Web applications when accessing databases.

Furthermore, the Web application server provides transaction support through an implementation of Java Transaction Server (JTS) in relation with JDBC/XA databases.

Finally, the IBM WebSphere Application Server Standard Edition architecture includes a system management infrastructure with two primary components the Administrative server and the Administrative client (console) (seeFigure 17 on page 36).

For further information on the system management structure see 2.1.5, "System Management Infrastructure" on page 40.

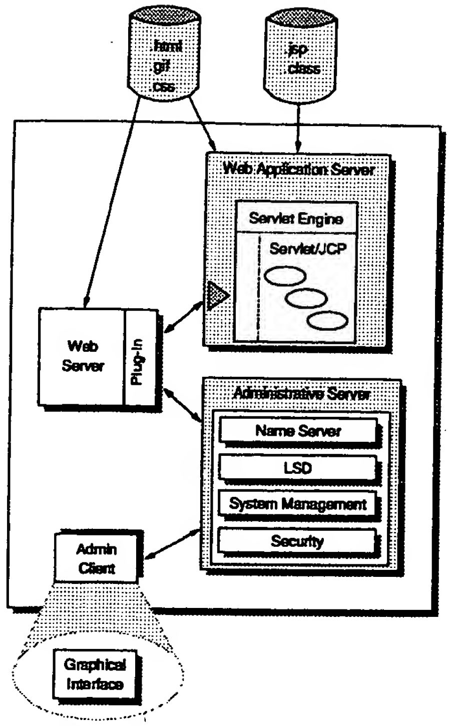


Figure 17. WebSphere Application Server Standard Edition 3.0 runtime architecture

2.1.1.3 WebSphere Application Server Advanced Edition

The WebSphere Application Server Advanced Edition (AE) has the functions found in the WebSphere Application Server Standard Edition plus support for EJBs and distributed (clustered) systems. This application server is one of two EJB servers provided in IBM WebSphere Enterprise Edition.

Since IBM WebSphere Application Server Advanced Edition is an extension of the Standard Edition you can easily move an application developed for a server running Standard Edition to servers running Advanced Edition.

The Web application server in IBM WebSphere Application Server Advanced Edition (see Figure 18 on page 37) includes support for EJB containers (for the EJBs) besides the servlet engine (for the servlets and JSPs) also found in Standard Edition.

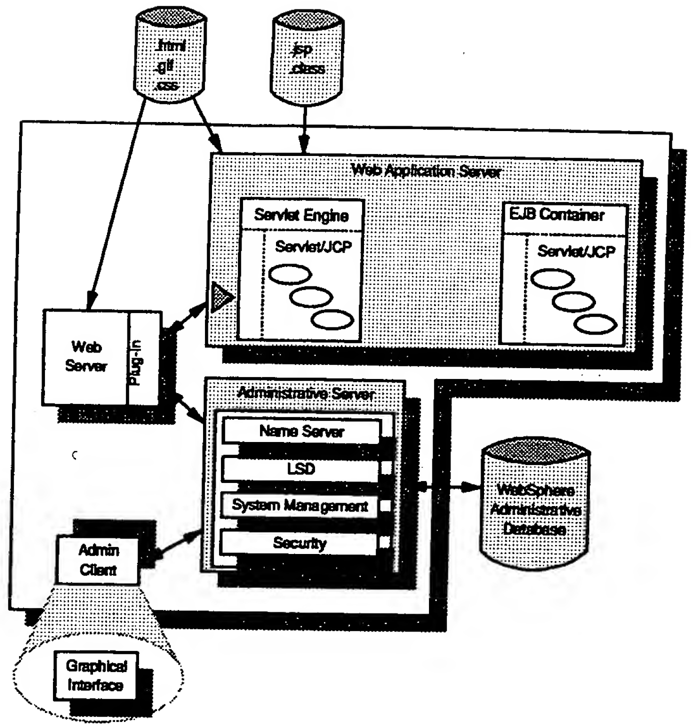


Figure 18. WebSphere Application Server Advanced Edition 3.0 runtime architecture

There is a single instance of the administrative server on a single node (physical machine).

In a multi-node setup (cell) there will be a single instance of the administrative server on each node. The administrative servers contains identical configuration information and access the same configuration repository in the same database (see Figure 19 on page 38).

The rationale is that you should be able to access any one of the administrative servers in a cell and see changes reflected on any other administrative server in the cell (single administrative image).

The WebSphere administrative database can be located either on a separate database server (physical machine) or on one of the machines that host the administrative servers. However, each machine must have database server, client or connection software installed and configured to enable access to the common database.

If you plan to run the same Web applications and enterprise applications on more than one physical machine you will also have to ensure that all systems have access to identical (or the same) files and in identical locations in the directory structures. This can be accomplished either by creating identical files and directory structures on each machine or by using a shared file system. If the first method is used we would recommend that you establish an automatic or semi-automatic procedure to ensure that the data are identical.

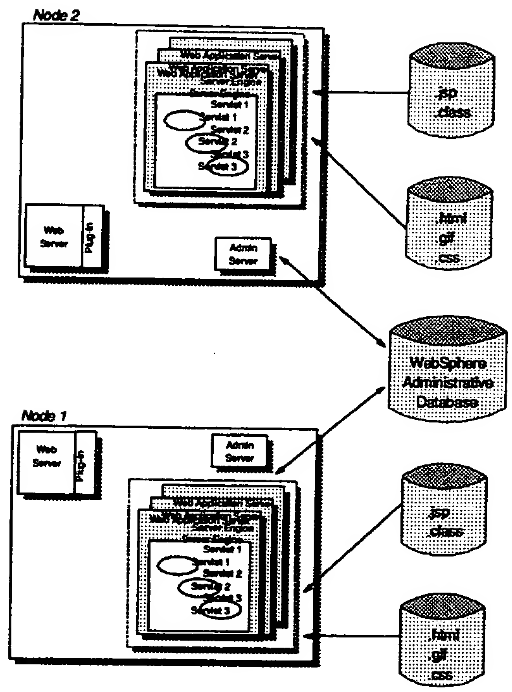


Figure 19. WebSphere Application Server Advanced Edition multi-node setup

A concept of server groups has been implemented in the IBM WebSphere Application Server for management and control purposes.

The following applies to the server group concept:

- Server groups consist of one or more Web application servers.
- A single Web application server can only belong to one server group.
- A server group can be seen as a logical server.
- Web application servers within a group are clones of each other. Clones in this context means logically identical application servers with respect to configuration of resources for example, containers and EJBs.

- The Web application servers within a server group may be distributed to different nodes (physical machines).

The server group is very useful in relation to workload management. A request can be forwarded to a server group and the request is handled by a server in the group while the specific server identity is hidden from the requester.

2.1.2 Enterprise Java beans and containers

The IBM WebSphere Application Server Version 3.0 server architecture provides a generic server (the Web application server, see Figure 18 on page 37) in which EJB containers live.

One of the primary responsibilities of an EJB container is to provide a number of fundamental services for example, transaction, state, security and persistence to the EJB. The advantage being that it reduces the work required by the EJB developers and supports development of portable EJBs.

Another responsibility of an EJB container is to create the interfaces (EJB Home and EJB Object) required for an EJB client to access a deployed EJB so the EJB client interacts indirectly with the EJB through the EJB container.

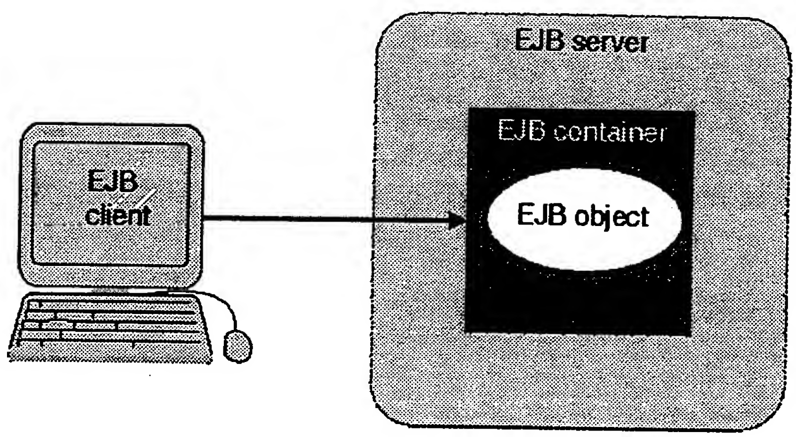


Figure 20. Client - EJB container relationship

2.1.3 Servlets and the application server

A special container (a servlet engine) is provided in IBM WebSphere Application Server to enable servlets to be run within the application server.

The servlet engine has been designed to be integrated as seamlessly as possible in the EJS architecture as well as in the security and system management infrastructure. The design approach allows for consistent system management for servlets and EJBs as appropriate while maintaining the typical servlet environment and characteristics intact.

Servlets are created and managed as members of a Web application in IBM WebSphere Application Server Version 3.0. IBM WebSphere Application Server provides work load management support for servlets (as it does for EJBs) to allow requests for servlets within a server group to be distributed to application servers belonging to the group.

2.1.4 Deployment tools

When you have developed your Enterprise Java beans for example, using VisualAge for Java or manually, they have to be deployed.

When you deploy an EJB you create or modify a Jar file which includes a description of the Jar file contents (the manifest), deployment descriptors, bean class files and potentially environment properties.

To create a deployed EJB Jar file you can for example, use VisualAge for Java, the Jetace tool or you can use the tools available in the Java Development Kit (JDK). The Jetace tool is provided as part of the IBM WebSphere Application Server.

After the EJB Jar file has been deployed for your IBM WebSphere Application Server it must be installed in your environment in accordance with the WebSphere administrative infrastructure. IBM WebSphere Application Server Advanced Edition provides the WebSphere Advanced Administrative Console to be used for EJB creation (installation).

Depending on the requirements for your EJB you may even choose to deploy an undeployed Jar file during EJB creation using the WebSphere Advanced Administrative Console.

You will also find a command line tool wlmjar with WebSphere Application Server Advanced Edition that can be used to create a workload-managed prepared version of your Jar file.

2.1.5 System Management Infrastructure

This section describes the IBM WebSphere Application Server administration model.

2.1.5.1 Systems management console

The WebSphere Administrative Console is the system management console for IBM WebSphere Application Server Version 3.0 Advanced Edition.

An administrative domain (sometimes referred to as a cell or sphere,) is a collection of managed nodes, that is, host machines. Each managed node has an administration server executing on that node that is responsible for configuring, monitoring, and managing the WebSphere servers that run on that node. A client graphical user interface is provided to enable the administrative domain to be defined.

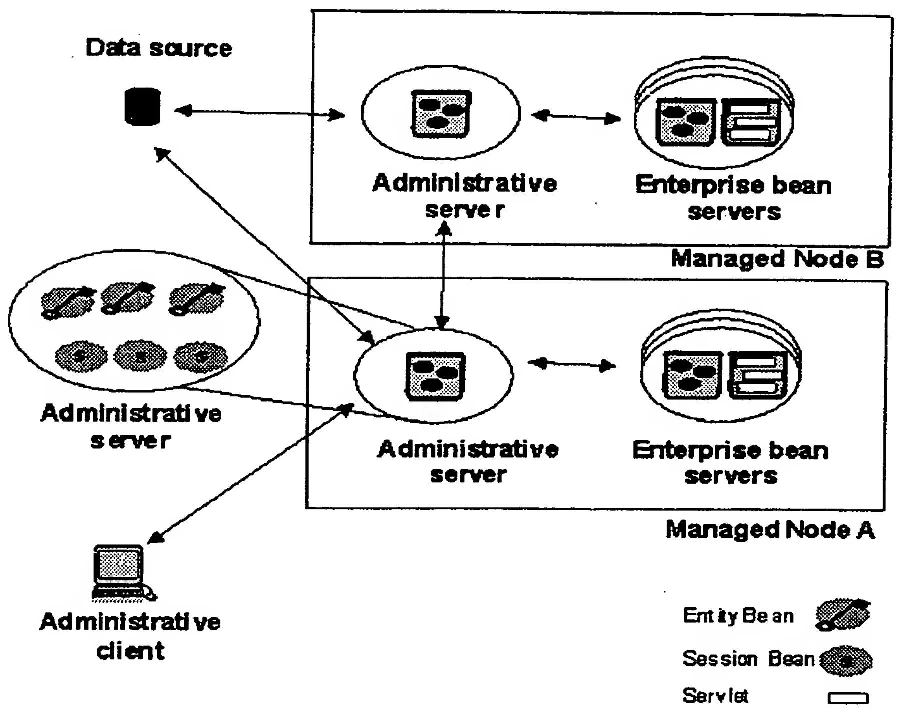


Figure 21. WebSphere Administration Server administration model

This Graphical User Interface (GUI) makes it easy to interact with the beans that were loaded by the adminserver. The front-end has a tab look and feel and has three main tabs as shown in Figure 22 on page 42, Figure 23 on page 43, and Figure 24 on page 45. The WebSphere Administrative Console

has three tab panes namely Types, Topology, and Tasks. These three panes are discussed below.

2.1.6 The Types view

The Types view is a hierarchical view of all resources on all nodes in the administrative domain. Each folder icon represents a different resource type. By selecting any object and right clicking, a context-sensitive menu appears. This menu has the basic tasks such as Create, Move, Default Properties.

The Types pane is shown in Figure 22 on page 42.

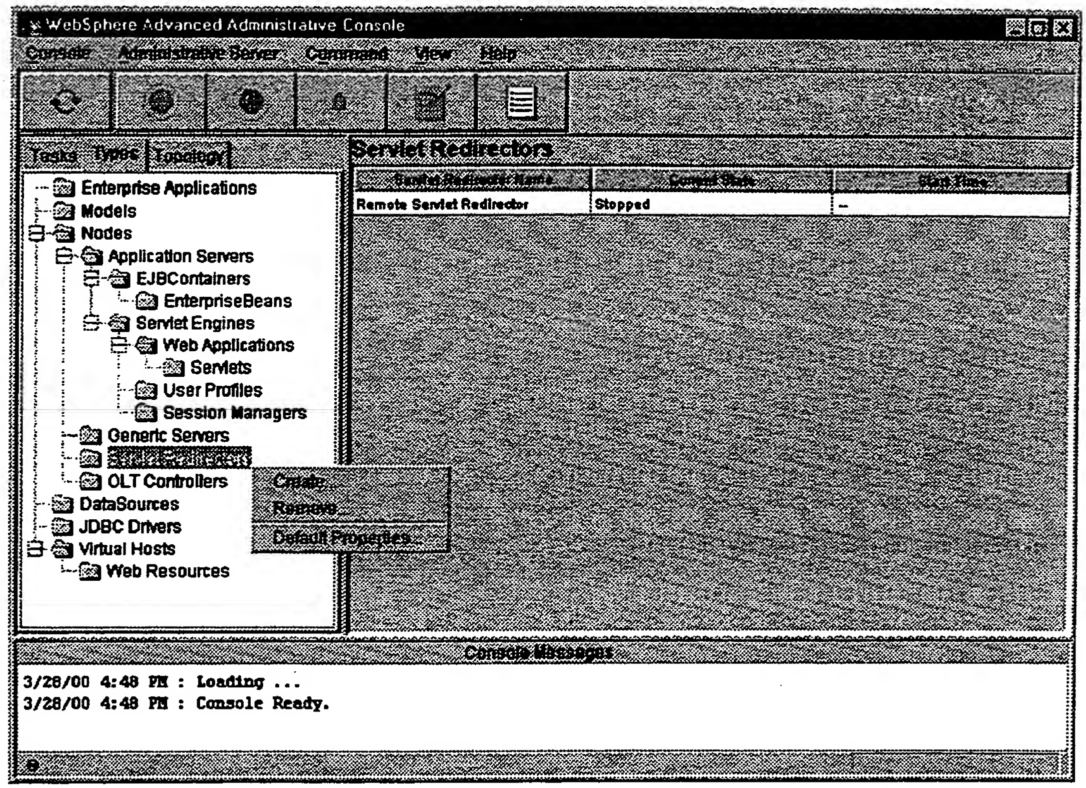


Figure 22. WebSphere Advanced Administrative Console - Types tab

2.1.7 The Topology view

The Topology pane consists of the WebSphere Administrative Domain and all the managed nodes as shown in Figure 23 on page 43. For each managed

node there is an associated hierarchy of all the resources within that node. The resource attributes can be changed and methods can be invoked on these resources in this view also, just as in the Types view.

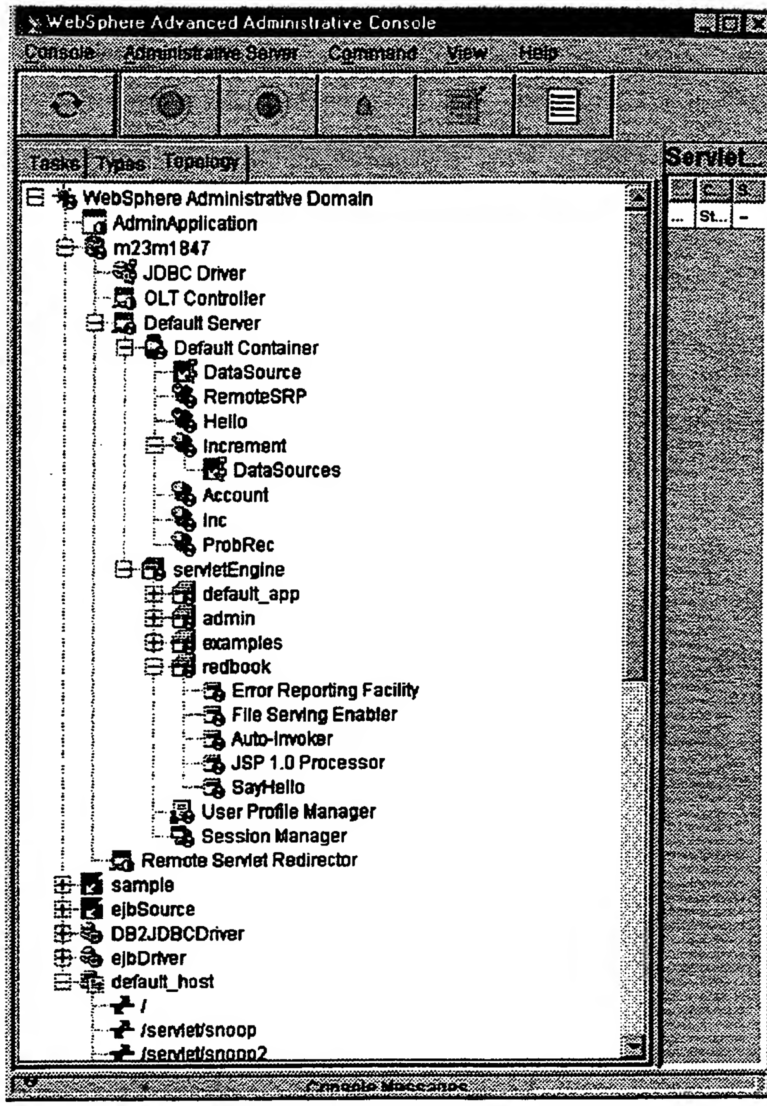


Figure 23. WebSphere Advanced Administrative Console - Topology tab

2.1.8 The Tasks view

The Tasks view shows different tasks that can be performed and is shown in Figure 24 on page 45. It consists of three task groups, which are briefly discussed below.

2.1.8.1 Configuration tasks

This group of tasks allows you to configure application servers, virtual hosts, servlet engines, web Applications, and enterprise applications.

2.1.8.2 Performance tasks

This task allows the user to launch the Resource Analyzer tool to monitor performance and load statistics. The Resource Analzer is dicussed in more detail in the redbook *WebSphere V3 Performance Tuning Guide*, SG24-5657-00.

2.1.8.3 Security tasks

Using this task, you can apply security to enterprise applications and to the HTTP and EJB methods of resources such as servlets and enterprise beans.

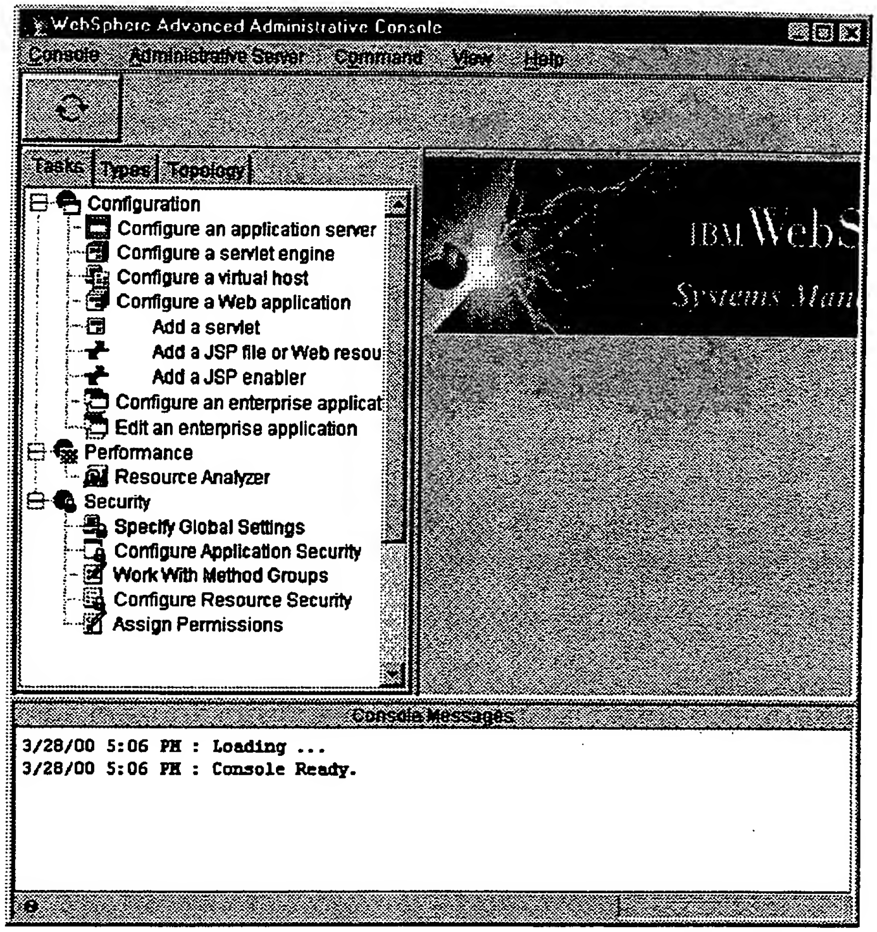


Figure 24. WebSphere Advanced Administrative Console - Tasks tab

8.1.6 Starting the LDAP directory server from the command line

To start LDAP from the command line you should use the following command:

```
/usr/ldap/bin/slapd
```

8.1.7 Administration interface - Web

From the Directory Server Web Admin you can perform several administrative task such as:

- Configure a database
- Configure a replica
- Start up and shut down the server
- Define an ACL
- Add and delete suffixes
- Add entries to the directory

To use the Directory Web Admin you have to start up a Web browser and type in the Location field `http://hostanme/ldap`.

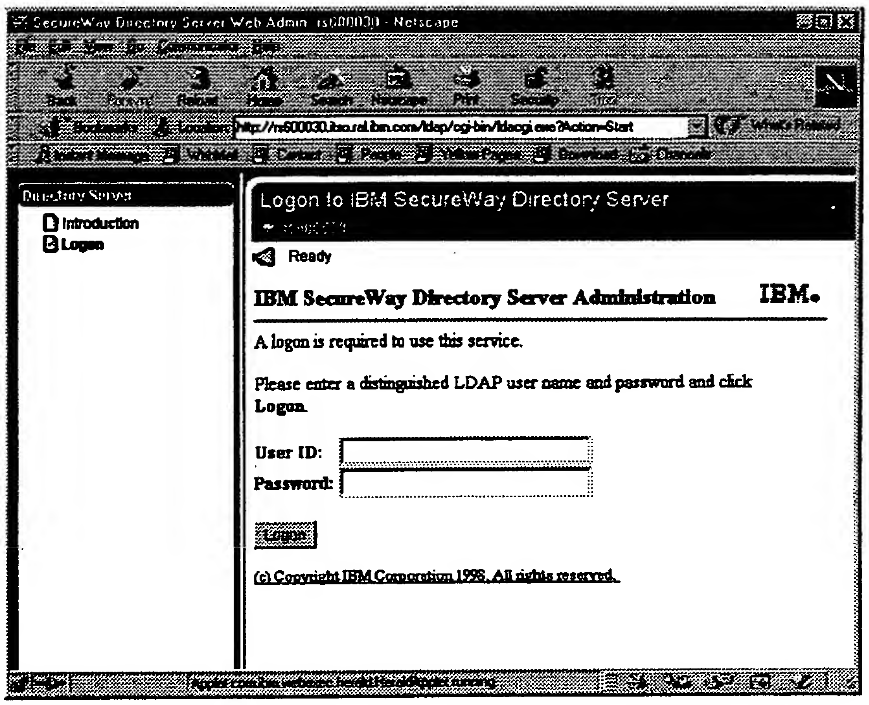


Figure 405. ldap Web admin

For the user ID specify cn=root or the user that you specified in the administrator DN and password section in Figure 401 on page 413.

For the password type the password that you specified in the administrator DN and password section and click the **Logon** button.

8.1.8 DMT interface

Use the DMT tool to:

- Connect to one or more directory servers via SSL or non-SSL connections.
- Display server properties and rebind to the server.
- List, add, edit, and delete schema attributes and object classes.
- List, add, edit, and delete directory entries.
- Modify directory entry ACLs.
- Search the directory tree.

You can configure the tool to automatically connect to one or more servers and to log in particular Distinguished Names (DNs) when it is started. To configure the tool, edit the /usr/ldap/etc/dmt.conf DMT configuration file. For example, the property file contains these lines:

- server1.url=ldap://localhost:389
- server1.security.bindDN=
- server1.security.password=
- server1.security.ssl.keyclass=

8.1.8.1 Log on to a server

If a directory user DN and password are not provided in the DMT configuration file, the tool connects as an anonymous user when it is started. Although an anonymous user can browse the directory tree and schema, to perform directory updates, in most instances, you need to log on as a directory user. To modify the directory server schema you must log on as the server administrator. To log on as a different user:

1. Click **Server -> Rebind**.
2. Provide the user DN and password.
3. Press Enter.

To start the DMT tool enter `dmt &`.

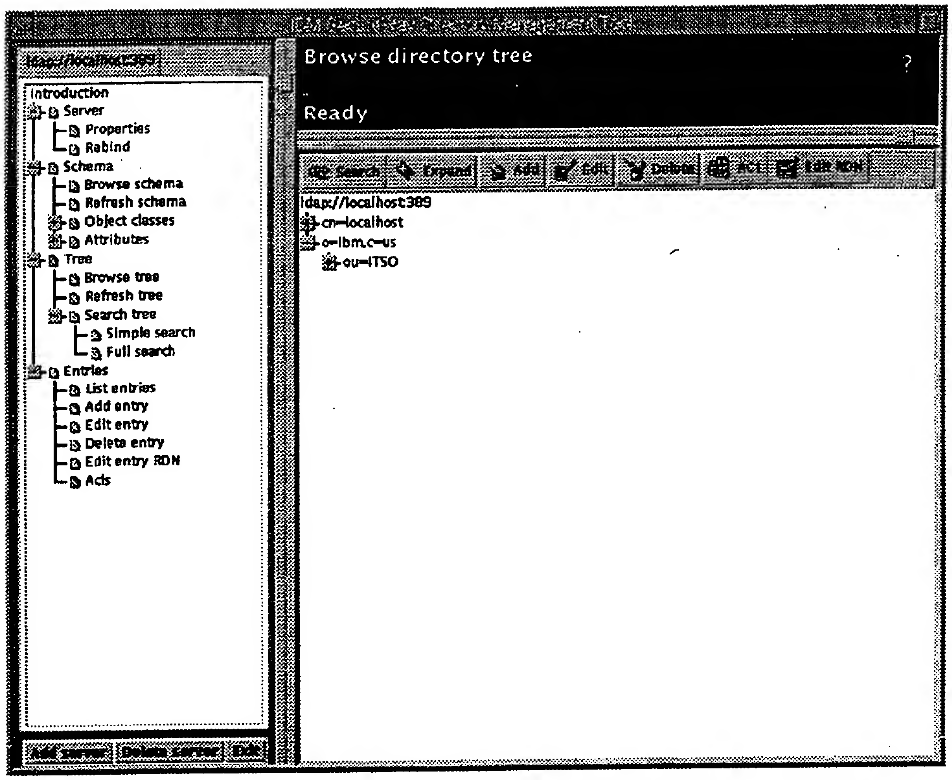


Figure 406. DMT

For more information about how to use this tool, see:

`http://www-4.ibm.com/software/network/directory/`

8.1.9 Configuration files - attributes and objects classes

The file /etc/slapd.conf contains configuration properties such as the SSL port, DB2 user, admin DN, and the admin password.

```
adminPW
>14II9OIR3WkRGgTEK+GHehiE31zkF1XEuvM3UXxkZ1Gm9KhP8FPcLPm4nBKB9fC2bRyCUARkQNikIPJa1NHS9a19KdYn3it
M4VjZxFmh/jFU2V0b7Bf2kTVvA+FSn1oyo8WVYolgU59+Z1jH458RneygaJdkz8iX2R<
adminDN   "cn=root"
# IBM SecureWay Directory Server Configuration File V3.1.1 for AIX
# CAUTION: EDIT THIS FILE WITH CARE
# We recommend making all changes through the server administration interface.
sysLogLevelm
```

```
errorLog/tmp/slapd.errors
includeSchema   /etc/ldapschema/V3.system.at
includeSchema   /etc/ldapschema/V3.ibm.at
includeSchema   /etc/ldapschema/V3.user.at
includeSchema   /etc/ldapschema/V3.system.oc
includeSchema   /etc/ldapschema/V3.ibm.oc
includeSchema   /etc/ldapschema/V3.user.oc
includeSchema   /etc/ldapschema/V3.ldapsyntaxes
includeSchema   /etc/ldapschema/V3.matchingrules
# The next file is for schema changes. It's empty when the package is installed
includeSchema   /etc/ldapschema/V3.modifiedschema
schemacheck V3_lenient
port389
securePort636
securitynone
#sslAuth options: serverauth/serverclientauth
sslAuth serverauth
sslCertificate  none
sslCipherSpecs  12288
sslKeyRingFile  key.kdb
sslKeyRingFilePWnone
maxthreads250
maxconnections240
waitingthreads  75
terminateIdleConnections on
timelimit900
sizelimit500
#pwencryption options: imask/none/crypt/SHA
pwencryption    imask
###########################################################################
# ldcf database definitions
###########################################################################
database        ldcf
suffix          "cn=schema"
###########################################################################
# rdbm database definitions
###########################################################################
databaserdbm
plugin          database /lib/libback-rdbm.a rdbm_backend_init
suffix"o=IBM,c=US"
databaseNameldapdb2
dbInstanceldapdb2
dbuserpw
>148NboL9SSFc2/qrSVKLTaT3j20tSLN3yzRp06VctGQpa15BX5Y6EFC+8RqNSw5mrTyCUARkQNikIPJa1NHS9a19KdYn3it
M7TiGOO4KTSe5NEENW1uvN23qWLcdT8i5bsWVYolgU59+Z1jH458RneygaJdkz8iX2R<
dbuseridldapdb2
dbConnections   6
suffix "cn=localhost"
readOnlyoff
# point this path at your changelog conf file and uncomment
# (it must be full path)
#include /etc/ldapschema/slapd.cl.conf
```

If you want to add a new object, modify or delete classes or attributes you can use the DMT tool. This tool also lets you know how the standard object

classes are created. Any attribute or class modification dynamically changes the directory schema.

To add an object click **Schema -> Object Classes -> Add Object class**.

To add an attribute click **Schema -> Attributes -> Add attribute.**

For further information see the SecureWay Directory Web site at: `http://www-4.ibm.com/software/network/directory/`.

8.1.10 LDAP commands

We now show you how to use some of the more important LDAP commands from the user's point of view. There are more specific IBM SecureWay LDAP commands for administrative purposes. Other vendors who have implemented these commands may be using different flags.

8.1.10.1 LDIF format

LDAP Data Interchange Format (LDIF) is a format for representing LDAP entries in text form. It is widely used and accepted as a de-facto standard. LDIF is used by the ldapmodify, ldapadd, and ldapsearch command-line utilities. The basic form for an entry in an LDIF file is as follows:

- dn: <distinguished name>
- objectClass: <object class>
- objectClass: <object class>
- <attrtype>: <attrvalue
- <attrtype>: <attrvalue>

An LDIF file consists of a series of records separated by a blank line. A record is a directory entry with a mandatory DN and at least an objectClass if the record is a new entry. If the record is for an update only, the DN is required. Some attributes, required by an objectClass, must be defined. Attribute values can be clear text, such as a name, or they can be Base64 encoded binary data, such as for a JPEG picture.

8.1.10.2 The ldapmodify and ldapadd utilities

The ldapmodify utility is a command-line utility built around the ldap_modify() API. The utility opens a connection to an LDAP server, binds to the server, and modifies an entry. The entry information is read from standard input or from a file. The ldapadd utility is implemented as a renamed version of ldapmodify. The ldapadd works the same way as the ldapmodify with the -a flag set. The syntax of the utility is:

```
ldapmodify [options] [-f <ldif input file>]

ldapadd [options] [-f <ldif input file>]
```

Some options are:

- -h host LDAP server host name
- -p port LDAP server port number, default 389
- -D dnbind dn by default anonymous
- -w bind password
- -R specifies that referrals are not to be automatically followed
- -M manage referral objects as normal entries
- -V LDAP protocol version (2 or 3; default is 3)
- -C charset character set name to use, as registered with IANA
- -b Assumes that any values that start with a / are binary values, and that the actual value is in a file whose path is specified in the place where values normally appear
- -c continuous operation; do not stop processing on error
- -n show what would be done but don't actually do it
- -v verbose mode
- -d level set debug level in LDAP library

Examples

Following are some examples using the LDAP commands:

```
ldapadd -h rs600030 -D "cn=root" -w swall7r -f add.ldif
```

This is the add.ldif's contents:

```
cn=Marisa,ou=ITSO,o=ibm,c=us
sn=cicsMan
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
uid=Marisa
mail=Marisa@ar.ibm.com
cn=Marisa
userPassword=swall7r
```

Now let's modify the Marisa's mail attribute:

```
ldapmodify -h rs600030 -D "cn=root" -w swall7r -d mod.ldif
```

```
cn=Marisa,ou=ITSO,o=ibm,c=us
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
mail=cicsMan@ar.ibm.com
```

To validate the change we ran the ldapsearch command:

```
ldapsearch -h rs600030 -b "ou=ITSO,o=IBM,c=US" cn=Marisa
```

```
cn=Marisa,ou=ITSO,o=ibm,c=us
sn=cicsMan
uid=Marisa
cn=Marisa
objectclass=ePerson
objectclass=person
objectclass=inetOrgPerson
objectclass=top
objectclass=organizationalPerson
mail=cicsMan@ar.ibm.com
```

8.1.10.3 The ldapsearch utility

The `ldapsearch` utility is a command-line utility built around the ldap_search() API. The utility opens a connection to an LDAP server, binds to the server, and performs a search using a specified search filter. If the request finds one or more entries, the requested attributes are retrieved, and the entries and values are printed to standard output. If no attributes are specified, all attributes associated with each returned entry are displayed. The syntax is:

```
ldapsearch [options] filter [attributes]
```

The significant parameter options for the ldapsearch tool are:

- -h LDAP server host name
- -p LDAP server port number
- -D bind dn
- -w bind password
- -b base dn for search; LDAP_BASEDN in environment is default
- -s search scope (base, one, or sub)

- -a how to reference aliases (never, always, search, or find)
- -l time limit (in seconds) for search
- -z size limit (in entries) for search
- -f perform sequence of searches using filters in file
- -A retrieve attribute names only (no values)
- -R do not automatically chase referrals
- -M manage referral objects as normal entries
- -V LDAP protocol version (2 or 3; default is 3)
- -C character set name to use, as registered with IANA
- -B do not suppress printing of non-ASCII values
- -L print entries in LDIF format (-B is implied)
- -F print separation between attribute names and values
- -t write values to files in /tmp
- -n show what would be done but don't actually do it
- -v run in verbose mode
- -d set debug level to level in LDAP library

The search filter, in many cases, will either be a simple attribute search (such as cn=Smith) or for all attributes (cn=*). Search filters, however, can be fairly complex, and there is a separate RFC (RFC 2254) that you should refer to if you need all the details. The following is a brief description of search filters. A search filter defines criteria that an entry must match to be returned from a search. The basic component of a search filter is an attribute value assertion of the form:

```
attribute operator value
```

For example, to search for a person named John Smith, the search filter would be cn=John Smith. In this case, cn is the attribute, = is the operator, and John Smith is the value. This search filter matches entries with the common name John Smith. Table 9 on page 424lists the operators for search filters.

Table 9. Operators

Operator	Description	Example
=	Returns entries whose attribute is equal to the value.	cn=John Smith finds the entry with the common name John Smith.
>=	Returns entries whose attribute is greater than or equal to the value.	sn>=smith finds all entries from smith to z*.
<=	Returns entries whose attribute is less than or equal to the value.	sn<=smith finds all entries from a* to smith.
=*	Returns entries that have a value set for that attribute.	sn=* finds all entries that have the sn attribute.
~=	Returns entries whose attribute value approximately matches the specified value. Typically, this is an algorithm that matches words that sound alike.	sn~= smit might find the entry "sn=smith".

The "*" character matches any substring and can be used with the = operator. For example, cn=J*Smi* would match John Smith and Jan Smitty. Search filters can be combined with Boolean operators to form more complex search filters. The syntax for combining search filters is:

("&" or "|" (filter1) (filter2) (filter3) ...)

("!" (filter))

The Boolean operators are listed in the following table:

Table 10. Operators

Boolean operator	Description
&	Returns entries matching all specified filter criteria.
|	Returns entries matching one or more of the filter criteria.
!	Returns entries for which the filter is not true. This operator can only be applied to a single filter. `(!(filter))` is valid, but `(!(filter1)(filter2))` is not.

Examples:

1. Retrieve all the entries with a person object defined;

```
ldapsearch -h rs600030 -b "o=IBM,c=US" objectclass=person
```

2. Retrieve all the entries with an e-mail ending in "ibm.com";

```
ldapsearch -h rs600030 -b "o=IBM,c=US" mail=*ibm.com
```

3. Retrieve all the entries with cn=Marina and mail=*ibm.com;

```
ldapsearch -h rs600030 -b "o=IBM,c=US"
"(&(cn=Marisa)(mail=*ibm.com))"
```

Note that we used " " in the filter. The reason was that on AIX the ksh preprocesses the () directive with " " and doesn't preprocess the contents.

4. Retrieve all entries with mail=*ibm.com and cn not equal to Karina, root;

```
ldapsearch -h rs600030 -b "o=IBM,c=US"
"(&(mail=*ibm.com)(!(|(cn=Karina)(cn=root))))"
```

8.1.10.4 The ldapdelete utility

The ldapdelete utility is built around the ldap_delete() API. The utility opens a connection to an LDAP server, binds to the server, and deletes one or more entries. The distinguished names (DNs) of the entries to delete are read from standard input or from a file.

The syntax is:

```
ldapdelete [options] [DNs]

ldapdelete [options] [-f file]
```

where:

`dn`: one or more items to delete

`file`: name of input file containing items to delete

If neither `dn` or `file` is specified then items are read from standard input. The options are:

-h LDAP server host name

-p LDAP server port number

-D bind dn

-w bind password

-Z use a secure ldap connection (SSL)

-K file to use for keys

-P keyfile password

-N private key name to use in keyfile

-m perform SASL bind with the given mechanism

-R do not chase referrals

-M Manage referral objects as normal entries

-O maximum number of referrals to follow in a sequence

-V LDAP protocol version (2 or 3; default is 3)

-C character set name to use, as registered with IANA

-c continuous operation; do not stop processing on error

-n show what would be done but don't actually do it

-v verbose mode

-d level (set debug level in LDAP library

Examples:

1. Delete the entries with cn=Karina:

```
ldapdelete -h rs600030 -D "cn=root" -w swal17r "cn=Karina,
ou=ITSO,o=IBM,c=US"
```

2. Delete two entries from a file called ldapDelete:

```
"cn=Marisa,ou=ITSO,o=IBM,c=US"
"cn=Cristian,ou=ITSO,o=IBM,c=US"
```

```
ldapdelete -h rs600030 -D swal17r -f ldapDelete
```

8.1.11 Configuring the Netscape address book to use LDAP

The Netscape address book is an LDAP-enabled application, which means that it can get information from an LDAP directory. Other products like Microsoft Internet Explorer are LDAP enabled too.

In these examples we show you how to set up the Netscape address book to use IBM SecureWay Directory V3.1.1.

1. Start the Netscape address book.

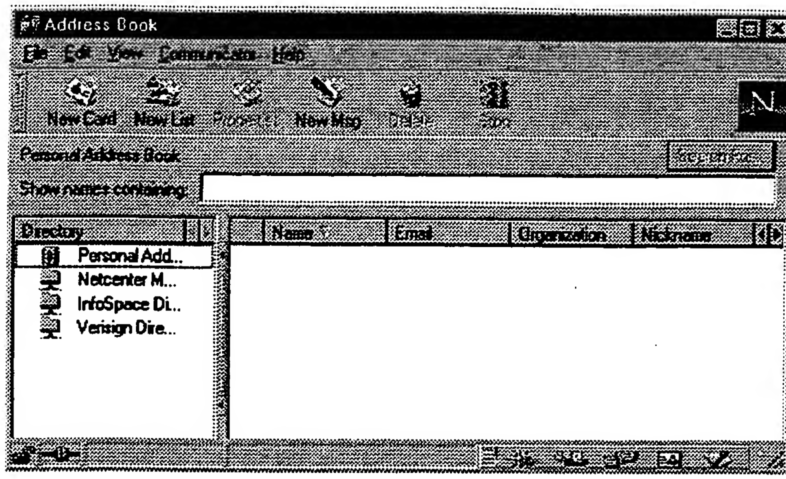


Figure 407. Netscape address book

2. Create a new directory.

Click **File -> New Directory** and you should see the dialog box shown in Figure 408.

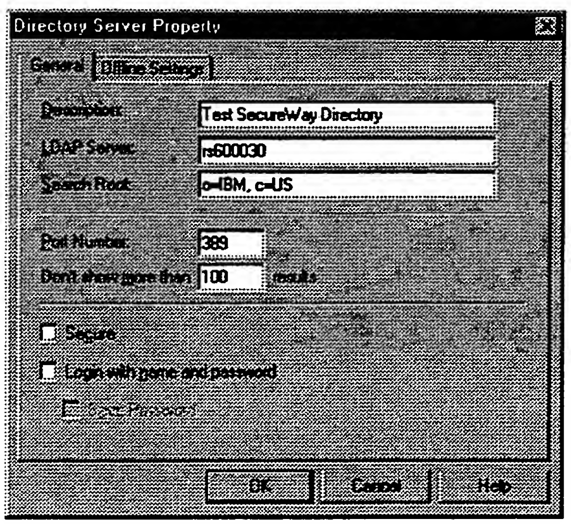


Figure 408. Directory Server Property dialog box

The fields are:

- Description: Title description is optional
- LDAP Server: Host name of the Directory Server
- Search Root: Base DN
- Port Number: By default 389
- Don't show more than: Maximum number of entries returned
- Secure: Use SSL
- Login with name and password: Used to bind with user ID and password

3. To see all entries, type * in the Show names containing field:

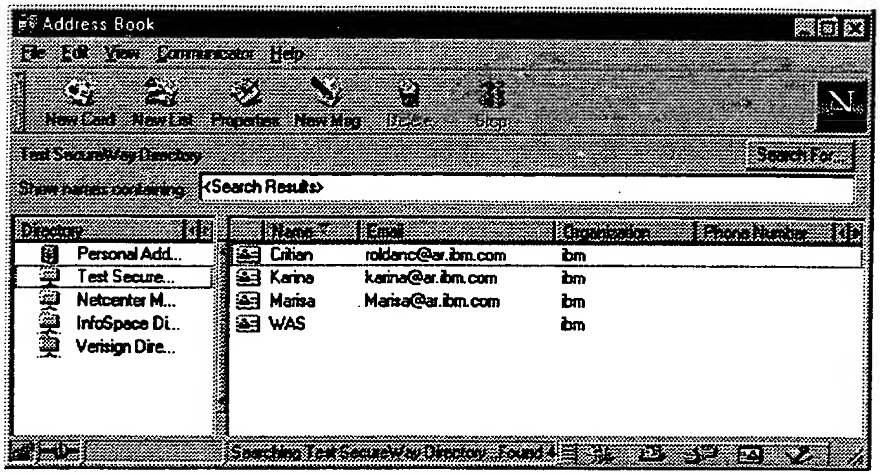


Figure 409. Search

8.1.12 WebSphere and LDAP

WebSphere supports authentication mechanisms based on validating credentials such as user ID and password, certificates, or tokens. Credentials are verified against a user registry supporting such a schema. For example, user IDs and password-based authentication can be based on the LDAP user registry where authentication is performed using an LDAP bind. A certificate validation list may be used in cases where authentication of the user is based on the client certificate presented by the user over a mutual SSL connection. WebSphere supports a three-party authentication schema, one in which the client principal and server principal are authenticated to a mutually trusted third-party. The third-party in this case is the authentication token server. An advantage of a three-party schema is that administration of the user registry can be controlled centrally.

WebSphere supports the following list of LDAP Directory Servers:

- Netscape Directory Server Version 3.x and 4.x
- SecureWay Directory Server Version 2.1 and 3.1.1
- Lotus Domino Version 4.6 and 5.0

Additional attributes are available for customizing any of the default filters to fit a Directory Server not listed above.

8.1.13 Configuring WebSphere to use LDAP

Start the WebSphere Advanced Administrative Console.

1. Select the **Tasks** tab, double-click the **Security** option and you will see a window similar to Figure 410:

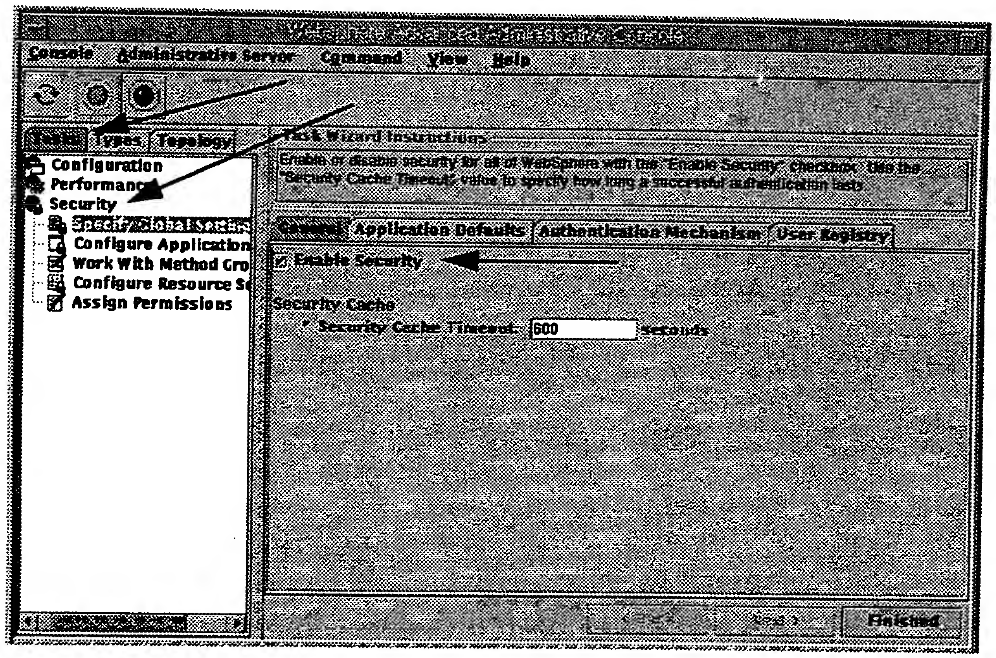


Figure 410. Security global settings

2. Select **Specify Global Settings** and click the **Enable Security** field.

   Enable Security: Specifies whether to enable or turn off server security. If you deselect this field all the security options specified on resources will be unprotected.

Security Cache TimeOut: Specifies how many seconds the server should cache security information received from the user registry (Operating System registry or Directory Server "LDAP").

3. Select the **Application Defaults** tab to specify the following properties:

   Realm Name: Is the security domain where the user will be authenticated? If the principal tries to access a resource in a different realm, the principal will be prompted to log in to the new realm. It is used if Single Sign On (SSO) is used.

   Challenge Type: This option specifies the mechanism used by the principal to interchange credentials. If you select **Basic**, it means that the Web browser will prompt the principal for a user ID and password.

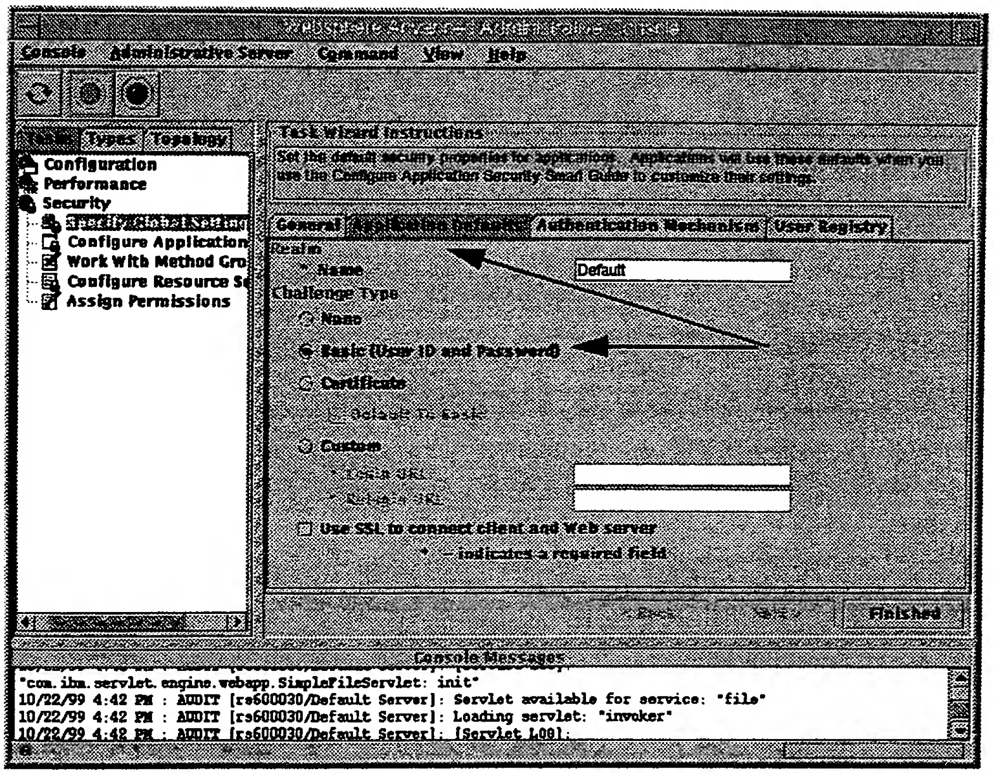


Figure 411. Global settings - Applications Defaults

4. Click the **Authentication Mechanism** tab. On this page you specify the mechanism used by the security server to authenticate the principal's credentials. See Figure 412 on page 431.

Lightweight Third Party Authentication (LTPA): Select this option to use the LDAP directory server as the registry system.

Token Expiration: Specifies how many minutes can pass before a client using an LTPA token must be authenticated again. We used the default value.

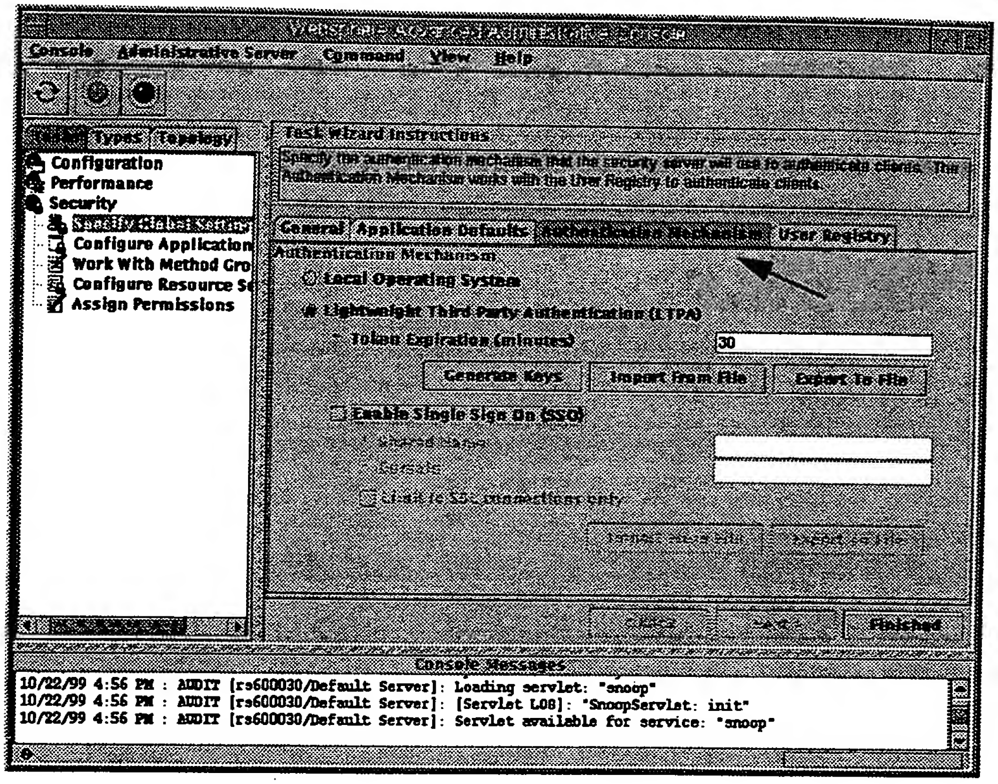


Figure 412. Global settings - Authentication Mechanism

5. Click the **User Registry** tab. On this page we specify basic information to use the LDAP Directory Server as shown in Figure 413 on page 433. These are the attributes that you should change:

   Security Server ID: Type a valid user ID registered in the LDAP Directory server used by the Application Server V3 Security server. The user ID should have some administrative privileges.

   Security Server Password: Type the password for the security server ID user.

Directory Type: Select the directory to be used. In our case we used SecureWay.

Host: Type the host name where the LDAP directory is running. In this example it is rs600030.

Port: Specify the LDAP directory port, which by default is 389. We used the default value.

Base Distinguished Name: Use this property to specify the starting point for LDAP searches. If you plan to use Domino directory you can leave this field blank. In this example we used ou=ITSO,o=IBM,c=US.

Bind Distinguished Name: Use this property to specify the DN to be used for the application server. If you leave it blank the Application Server binds using the anonymous ID. In this field we used the LDAP administrator cn=root.

Bind Password: Use this field to specify the password used to bind to the LDAP directory. We used the LDAP administrator's password.

Note: If you get the exception org.omg.CORBA.TRANSACTION_ROLLEDBACK when you click the Finished button, it means that you probably specified an incorrect user ID or password.

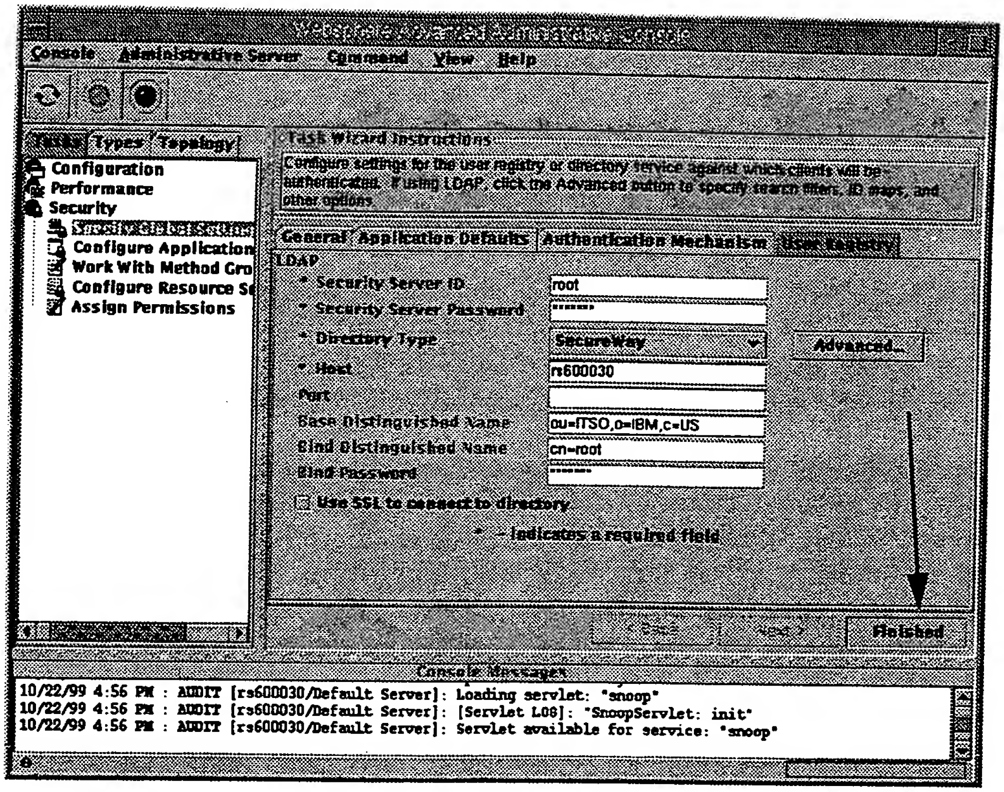


Figure 413. User Registry properties

6. Click the **Finished** button.

   Note: For any change made on the global security settings you must restart the node. To do so, click **Topology -> WebSphere Admin Domain -> Host Name (rs600030)**. Then right-click the mouse button and select **stop** or **restart**. You must leave the administrative console.

You can customize more LDAP attributes such as search filters and certificate mapping as shown in Figure 414:

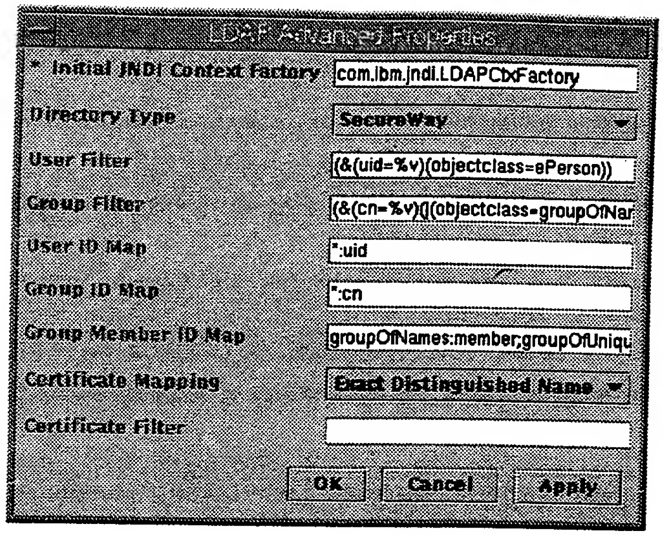


Figure 414. Advanced properties

7. The next step consists of applied security on an application. This means performing the following steps:
   a. Configure an enterprise application.
   b. Configure application security.
   c. Configure resource security.
   d. Assign permissions.

   For more information about how to perform these tasks see the WebSphere documentation available in the directory /usr/WebSphere/AppServer/web/doc/begin_here/index.html

8.1.14 JNDI

JNDI defined by Sun Microsystems provides naming and directory functions to Java programs. JNDI is an API independent of any specific directory service implementation.

The definition prevents, by design, the appearance of any implementation-specific artifacts in the API. The API is designed to cover the common case. JNDI was developed as part of Java Enterprise API set which also includes Enterprise Java beans (EJB) and Java Database Connectivity (JDBC). The EJB specification has a special relationship with JNDI because EJB uses this mechanism to find Entity beans or Sessions beans.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☑ BLACK BORDERS
- ☑ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☑ FADED TEXT OR DRAWING
- ☑ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☑ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☑ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: ______________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.